In Some Curved Spaces, One Can Solve NP-Hard Problems in Polynomial Time

Vladik Kreinovich[1] and Maurice Margenstern[2]
[1]Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
vladik@utep.edu
[2]Université Paul Verlaine – Metz
UFR MIM, LITA, EA 3097
Ile du Saulcy 57045 Metz
Cédex France margens@univ-metz.fr

Abstract

In the late 1970s and the early 1980s, Yuri Matiyasevich actively used his knowledge of engineering and physical phenomena to come up with parallelized schemes for solving NP-hard problems in polynomial time. In this paper, we describe one such scheme in which we use parallel computation in curved spaces.

1 Introduction and Formulation of the Problem

Many practical problems are NP-hard. It is well known that many important practical problems are NP-hard; see, e.g., [11, 14, 27]. Under the usual hypothesis that P≠NP, NP-hardness has the following intuitive meaning: every algorithm which solves all instances of the corresponding problem requires, for some instances, non-realistic hyper-polynomial (probably exponential) time on a Turing machine (and thus, on most known computational devices).

How can we solve NP-hard problems? Matiyasevich's dream. The difficulty with NP-hard practical problems is that their solution on the existing computational devices requires an unrealistically large time.

Meanwhile, computers are getting faster and faster. The main reason for this speed-up is that computer designers are constantly incorporating new ideas from engineering and physics. It is therefore reasonable to look for new ideas from engineering and physics, ideas which would enable us to solve NP-hard problems in feasible (polynomial) time. After the main theory of NP-hardness

was developed in the early 1970s, several researchers have tried to find such ideas. One of the first pioneers in this research direction was Yuri Matiyasevich.

In late 1970s and early 1980s, Matiyasevich actively used his knowledge of engineering and physical phenomena to come up with parallelized schemes for solving NP-hard problems in polynomial time. He presented several related ideas in his talks and papers; see, e.g., [38, 39]. Some of the technical and physics ideas on which he worked at that time (based on super-conductivity, non-traditional chemical reactions, and other unusual physical and engineering phenomena) did not yet lead to promising breakthroughs; however, other ideas can be viewed as early predecessors of successful techniques such as DNA computing; see, e.g., [1, 2].

Overall, his actively pursued dream of solving NP-hard problems in reasonable time has encouraged many researchers to come up with other ideas which led to practically useful algorithms. For example, Matiyasevich's pursuit of such a speedup was one of the main inspirations behind psychology-motivated semi-heuristic fast algorithms for solving NP-hard problems which were developed by S. Maslov in the early 1980s and described in [38]; see also [25].

In this paper, we will follow a different idea: the use of parallel computation in curved spaces to speed up the solution of NP-hard problems.

Parallelism is not a panacea. When running an algorithm takes too much time on a single computer, a natural idea is to run several computers in parallel. At first glance, it may seem possible that adding a large number of processors can enable us to run a previously non-feasible algorithm in a reasonably short time. However, there are geometric arguments that although we can drastically decrease the computation time by using parallel processors, we cannot make a non-feasible algorithm feasible; see, e.g., [12, 40]. Let us present these arguments.

Indeed, according to modern physics, the fastest speed of an arbitrary process is the speed of light c. So, if a parallel computer finishes its computations in time t and it contains a processor which is located at a distance $r > c \cdot t$ from the user, then this processor will not influence the result of the algorithm: even when traveling with the speed of light, the signals from this processor will reach the user only *after* the time $r/c > t$ (i.e., after the computations are over). Therefore, all processors that participate in this computation are located at a distance $r \le c \cdot t$ from the user, i.e., inside a sphere of radius $r = c \cdot t$ with a center at the location L of the user.

On each technological level, there is a lower bound V_0 on the volume of a processor. So, within a sphere of radius r, we can fit no more than $N \stackrel{\text{def}}{=} V(r)/V_0$ processors, where $V(r)$ denotes the volume of a sphere of radius r with the center at the point L. In the Euclidean space, $V(r) = (4/3) \cdot \pi \cdot r^3$, so, with $r = c \cdot t$, we can fit no more than $C \cdot t^3$ processors, where $C \stackrel{\text{def}}{=} (4/3) \cdot \pi \cdot (c^3/V_0)$. We can simulate this parallel computation on a sequential machine: first, we simulate what each of N processors does in the first time unit, then what they all do in the second time unit, etc.

To simulate each time unit of a parallel machine, we thus need N time units on a sequential machine. So, the algorithm that took time t on a parallel machine with N processors, can be simulated on a sequential machine in time $T = N \cdot t$. Since $N \le C \cdot t^3$, we thus have $T \le C \cdot t^4$, hence $t \ge C^{1/4} \cdot T^{1/4}$.

So, if a problem requires exponential time T on a sequential computer, it will still require exponential time t (smaller but still exponential) on a parallel computer. Similarly, if a problem can be resolved in polynomial time t on a parallel computer, then it can also be solved in a polynomial time T (larger but still polynomial) on a sequential computer.

In short, no matter how many processors we have, parallelism does not automatically lead to a possibility of solving NP-hard problems in polynomial time.

Space-time curvature can help to speed up parallel computations: what is known. According to modern physics, real space-time is not Euclidean, it is curved. Because of this curvature, the dependence $V(r)$ of the volume of the sphere on its radius r is, in general, different from the Euclidean formula $V(r) \sim r^3$.

In particular, in a hyperbolic (Lobachevsky) space, historically the first non-Euclidean space, the volume $V(r)$ grows exponentially with the radius r; see, e.g., [7]. A similar exponential growth happens for other more realistic physical spaces; see, e.g., [40]. In our papers, we have shown that this exponential growth can be actually used to speed up parallel computations. Namely, we can fit an exponential number of processors within the given distance from the user and thus, we can check an exponential number of cases in feasible time. For example, to solve a propositional satisfiability problem with n Boolean variables $x_1, \ldots, x_n$ (the classical NP-hard problem), we can have 2^n processors each of which is assigned an n-dimensional Boolean vector; these processors, in parallel, check whether a given propositional formula F holds for the corresponding vectors.

In [40], the resulting speed up was presented as a theoretical possibility, based on the fact that, e.g., in a hyperbolic space, we can fit an exponential number ($\sim \exp(\alpha \cdot r)$) of non-intersecting spheres of radius r_0 into a sphere of radius r. In that paper, we did not present any explicit scheme for placing the processors.

In [10, 15, 22, 28, 29, 30, 31, 32, 33, 34, 35, 36], explicit iterative geometric schemes are described that enable us to naturally fill the hyperbolic sphere of a large radius r with exponentially many small spheres of radius r_0.

Remaining problem. We have mentioned that there exist algorithms which use processors in curved spaces to solve NP-hard problems in polynomial time. These algorithms are based on the possibility to set up exponentially many processors within a sphere of a given radius.

However, these algorithms implicitly assume that these (exponentially many) processors are already there. In principle, we can set up these processors one

by one, but that setting up would require exponential time. As a result, with such sequential setting up, the overall time for setting up and computation will still be exponential.

So, the problem is how to parallelize the setting up of all these processors, so that the resulting overall time will be polynomial.

What we plan to do. In this paper, we describe such a scheme for parallel setting up in polynomial time.

2 Towards Proposed Solution: Informal Description of the Main Ideas

How to describe this problem in algorithmic terms: available techniques. The original Turing machines and similar computational devices (implicitly) assume that we already have infinitely many memory cells. It does not allow for designing and setting up of new cells and/or new processors.

Of course, we can always view an infinite tape of a Turing machine as only *potentially* infinite, in the sense that at any given time, we only have finitely many cells on this tape, and new cells are added when necessary. However, in this view, cells are added one by one – exactly what we wanted to avoid.

There is, however, a known extension of Turing machines which was specifically designed to take into account the possibility of adding new cells and/or new processors. This extension was originally proposed by Kolmogorov and Uspensky in the 1950s [23, 24]; overviews of the resulting Kolmogorov-Uspensky algorithms can be found, e.g., in [6, 17, 21, 45]. Such algorithms are known to have several advantages over more traditional Turing machine-type definitions: e.g., according to [6]: "Leonid Levin used a universal Kolmogorov machine to construct his algorithm for NP problems that is optimal up to a multiplicative constant [26]; see also [16]. The up-to-a-multiplicative-constant form is not believed to be achievable for the multi-tape Turing machine model popular in theoretical computer science."

Kolmogorov-Uspensky algorithms served as a basis for an even more general formalization of the abstract notion of computability: the notion of an Abstract State Machine [8, 17, 18, 19, 20, 21, 42].

What needs to be modified. In principle, Kolmogorov-Uspensky algorithms can be described in two versions:

- based on graphs in Euclidean space and
- based on abstract graphs.

The Euclidean space description is clearly not what we need.

In the abstract graph approach, we count each connection between vertices as 1 computational step. For example, in this version, we can build a full graph

of size 2^n in which every node is connected to every other node, with connection between every two nodes taking exactly 1 step. Of course, we can thus solve satisfiability problem in linear number of steps – but, as we have seen on the example of Euclidean space, the actual distance between the corresponding vertices can be exponential and so, one "step" in this sense may mean exponential time.

What we need is a definition specifically tailored towards a given curved space.

We need self-replication. At first glance, it may look like a generalization of Kolmogorov-Uspensky-type algorithms to curved spaces should be reasonably straightforward. From the purely *mathematical* viewpoint, it is indeed possible to provide a generalization. However, our goal is not simply to produce a mathematical generalization, but to provide a generalization which will be *physically* meaningful, a generalization in which 1 computational step indeed means 1 time unit (or at least constant number of time units).

Let us recall that what we want are processors which not only perform computations but also actually manufacture new processors. In other words, what we want are not simply processors, but rather intelligent robots.

Self-replicating programs and robots and physically possible. The possibility of a self-replicating automaton was first shown by John von Neumann [47] (see also [3, 4, 48]). Von Neumann's construction is very general: it can be used to design a self-replicating *computer program* or a self-replicating *robot*. Both possibilities are realistic. Self-replicating computer programs have been actually designed; in their malicious form, they are known as *computer viruses*. Simple self-replicating robots have also actually been designed; see, e.g., [50].

The potential of using self-replicating robots in space exploration has been analyzed since the 1970s. In honor of John von Neumann, such robots are called *von Neumann machines*, or *von Neumann probes*. The resulting scheme is as follows: a von Neumann probe, consisting of an interstellar propulsion system and a von Neumann replicator, is launched from the Earth toward a neighboring stellar system. Upon arrival it seeks out raw materials from local sources, and uses these to make several copies of itself (including its rocket engines). The copies are then launched at the next set of neighboring stars, etc. Technical details of this scheme are described in research papers [5, 9, 13, 46] and in a NASA Technical Report [49].

Difficulties. The above papers describe the design of self-replicating robots whose main objective is *space exploration*. Our objective is different: we want to use these robots to perform *parallel computations*.

Let us enumerate some potential difficulties related to this new objective. Both difficulties are related to the fact that these robots need to communicate with each other.

First difficulty: communicating at a large distance r may require exponentially growing energy. In the Euclidean space, it is very reasonable to assume that two robots can communicate at a distance $\le c \cdot n$ for some constant c, where n is the bit size of the input. Indeed, a signal sent by any source (in particular, a signal sent by a robot) spreads around. By the time this signal reaches distance r, the energy of the original signal is spread around the (surface of the) sphere of radius r, so its (average) energy density is equal to $E_0/S(r)$, where E_0 is the original energy and $S(r)$ is the area of the sphere of radius r. (In these back-of-the-envelope computations, we assume that the transmission is ideal and ignore possible signal decay; if we take this decay into account, the difficulties only become worse.)

The signal is detectable at a distance r if its density exceeds a certain detection threshold ρ_0: $E_0/S(r) \ge \rho_0$. Thus, to be able to transmit information to a a robot at distance r, we must generate the signal with the energy $E_0 \ge S(r) \cdot \rho_0$.

In the Euclidean space, $S(r) \sim r^2$, so the required energy increases polynomially with r – this is feasible to arrange, e.g., by combining a polynomial number of energy sources. However, we are interested in spaces in which the volume $V(r)$ grows exponentially and thus, the area $S(r)$ can also grow exponentially. In such spaces, to propagate the signal through distance r, we may need an energy which grows exponentially with r – i.e., an energy which is not feasible to implement.

How to avoid this first difficulty. To avoid this difficulty, we will assume that the robots can only communicate with each other when their distance is bounded by some constant d_0.

For this to be possible, we must make sure that by combining such bounded communications, we can set up a link between arbitrarily distant points x and x'. This is true in the Euclidean space, where we can connect x and x' by a straight line segment, and on that segment, set up points x_2, $\ldots$, x_{m-1} at distance d_0 from other another so that for $x_1 \stackrel{\text{def}}{=} x$ and $x_m \stackrel{\text{def}}{=} x'$, we have $d(x, x') = d(x_1, x_2) + \ldots + d(x_{m-1}, x_m)$ and $d(x_i, x_{i+1}) \le d_0$ for all i. A similar construction works in the hyperbolic space as well.

In our analysis, we will thus restrict ourselves to "d_0-connected" metric spaces, in which such a connecting sequence of d_0-close points exists for all x and x'. (A precise definition will be given in the next section.)

Second difficulty: number of communications can grow exponentially. Even when we solve the difficulty related to the weakness of signals, we still face the second difficulty: that there is a very large number of robots and thus, a very large number of communications.

Namely, as we have mentioned, the main idea behind the desired speed up is that all exponentially many processors perform computations in parallel, and then they send their results to the user. In the Euclidean space, we have at most polynomially many robots within a zone of polynomially growing radius,

and thus, we have at most polynomially many messages. We can process them sequentially or we can set up polynomially many receptors at the user's location.

In a curved space, we may have exponentially many robots and thus, exponentially many signals coming to the user. If we process these signals one by one, we will need exponential time. If we set up exponentially many receptors at the user's location, we will need exponentially large volume there – and we will still face a problem of sending signals from these receptors to the user.

How to avoid the second difficulty. To avoid this difficulty, in our algorithm, we will not send signals directly back to the user, we will try to collect them in stages, tree-like: several robots send their messages to a boss-robot, these boss-robots send to bosses of next level, etc., so that at each time unit, each robot receives $\le M_0$ messages for some fixed constant M_0.

We will thus restrict ourselves to spaces in which every robot can have no more than M_0 neighboring computing robots. Let us describe these properties in precise terms.

3 Proposed Solution: Definitions and the Main Result

First, we need a formal definition of appropriate metric spaces

For computational purposes, it is sufficient to consider discrete spaces. Physicists usually consider *continuous* spaces as models of physical reality. However, we are interested not in abstract points in space, but rather in points at which we can set up computational devices. At a given technological level, every computational device has a certain linear size $\varepsilon > 0$. This means that if we have already set up a device at some point x, then we cannot set up any other device at a distance $\le \varepsilon$ from x.

Thus, from the viewpoint of setting up computational devices, instead of considering all possible points, it is sufficient to consider only points distant from each other by no less than ε – e.g., points on a grid of size $\ge \varepsilon$. Thus, we arrive at the following definition.

Definition 1. *Let $\varepsilon > 0$ be a real number. We say that a metric space (X, d) is* ε-discrete *if for every two points $x \ne x'$, we have $d(x, x') \ge \varepsilon$.*

Comment. In the following text, we will only consider ε-discrete metric spaces.

For such spaces, it is natural to define a volume of a set simply as the number of its elements.

Definition 2. *Let (X, d) be a ε-discrete metric space, and $S \subseteq X$ be a subset of this space. By a* volume $V(S)$ *of this set S, we mean the number of elements in the set S.*

Comment. Our objective is to describe physical models in which we can perform exponentially many operations in polynomial time. For this to be possible, we must make sure that the volume of a ball $B_R(x_0) = \{x : d(x, x_0) \le R\}$ in this space grows exponentially with its radius R.

Definition 3. *We say that an ε-discrete metric space (X, d)* has exponential growth *if for some point x_0 and for some positive real numbers R_0, A, and k, for every $R \ge R_0$, the volume $V(B_R(x_0))$ of a ball $B_R(x_0) = \{x : d(x, x_0) \le R\}$ of radius R satisfies the inequality $V(B_R(x_0)) \ge A \cdot \exp(k \cdot R)$.*

Comment. In this definition, we assumed that the exponential growth property occurs for a single point x_0. It is worth mentioning that that once this property holds for one point x_0, it is true for every other point $x \in X$:

Proposition 1. *Let (X, d) have exponential growth. Then, for every point $x' \in X$, there exist positive real numbers R_0', A', and k', such that for every $R \ge R_0'$, the volume $V(B_R(x'))$ satisfies the inequality $V(B_R(x')) \ge A' \cdot \exp(k' \cdot R)$.*

Proof of Proposition 1. According to the triangle inequality, for every point x, we have $d(x, x') \le d(x, x_0) + d(x_0, x')$. Thus, for every radius $r > 0$, we have $B_r(x_0) \subseteq B_{r+d(x',x_0)}(x')$. In particular, for every $R \ge d(x', x_0)$, we thus have $B_R(x') \supseteq B_{R-d(x',x_0)}(x')$. Since the volume is simply defined as the number of points in a set, we therefore have $V(B_R(x')) \supseteq V(B_{R-d(x',x_0)}(x'))$.

If $R - d(x', x_0) \ge R_0$, i.e., if $R \ge R' \stackrel{\text{def}}{=} R_0 + d(x, x_0)$, then

$$V(B_R(x')) \supseteq V(B_{R-d(x',x_0)}(x')) \ge A \cdot \exp(k \cdot (R - d(x', x_0)) = A' \cdot \exp(-k \cdot R),$$

where $A' \stackrel{\text{def}}{=} A \cdot \exp(-k \cdot d(x', x_0))$. So, the desired property indeed holds for the above A', R_0', and $k' = k$. The proposition is proven.

Comment. To implement fast computations, it is not enough to know that the metric space has exponential growth. Indeed, we may have a metric space consisting of 2^n points in which every two points have the same distance $d \ge \varepsilon$. In this space, an arbitrary permutation preserves the distance. Since every two points can be swapped by an appropriate permutation, every two points in this space are equivalent to each other. So here, there are exponentially many points, but it is not clear how to distribute the task between all these points since they are all equivalent to each other. What helps for hyperbolic space (and for similar spaces described in [40]) is that in this space, neighborhoods of small radius are small, so in principle, we can cover them in parallel without losing exponential speed-up. Let us formalize the notion of such spaces.

Definition 4. *Let $d_0 > 0$ be a real number. We say that two points x and x' in a metric space (X, d) are* d_0-close *(or* d_0-neighbors*) if $d(x, x') \le d_0$.*

Comment. In what follows, in some cases when the value d_0 will be clear from the context, we will simplify the text by simply writing "neighbors" and "close".

Definition 5. *Let $d_0 > 0$ be a real number. We say that a metric space (X, d) is* d_0-connected *if for every two points $x, x' \in X$, there is a sequence $x_1 = x, x_2, \ldots, x_{m-1}, x_m = x'$ such that for every i, x_i and x_{i+1} are d_0-neighbors and*

$$d(x, x') = d(x_1, x_2) + d(x_2, x_3) + \ldots + d(x_{m-1}, x_m).$$

Definition 6. *Let $d_0 > 0$ be a given real number and $M_0 > 0$ be a given integer. We say that a metric space (X, d) has* M_0-bounded d_0-neighborhoods *if every point $x \in X$ has no more than M_0 d_0-neighbors.*

Definition 7. *Let $\varepsilon > 0$ and $d_0 > 0$ be real numbers, and $M_0 > 0$ be an integer. By an* potentially (ε, d_0, M_0)-computation enhancing space *(or simply* potentially computation-enhancing space*, for short), we mean an ε-discrete d_0-connected metric space with exponential growth and M_0-bounded d_0-neighborhoods.*

In what follows, we assume that a computation-enhancing space (X, d) is given, and that we also are given a point x_0 in this space. This point will be called the *user's location.*

The corresponding "robotic computer" will consist of "computational robots" located in different points of the potentially computation-enhancing space X. Let us start our definition of a robotic computer by defining the notion of a computing robot.

Definition 8. *By a* computing robot*, we mean a regular computer (e.g., a Turing machine or RAM, or a Java virtual machine) with a finite memory (sequence of bits) in which there are three additional commands:* build*,* send forward *(with memory location as a parameter), and* send backward *(with memory location as a parameter). The memory is divided into five parts:*

- *a* hardwired *part which contains a program;*
- *a* working *memory part which can be used for computations in the usual way;*
- *a* tree ID *memory part; it will be used for storing a special "tree ID" of a processor;*
- *a* regular ID *memory part; it will be used for storing a special "regular ID" of a processor;*
- *a* received *memory part; this received part is divided into M_0 subparts.*

We assume that the tree ID memory consists of a single bit (called a built *bit) followed by (one or several)* blocks *of bit size* $1 + B$*, where* $B \stackrel{\text{def}}{=} \lceil \log_2(M_0) \rceil$.

Comment. In a potentially computation-enhancing space (X, d), every location has $\leq M_0$ d_0-neighbors, so a B-bit long part of the block is sufficient to differentiate between all the neighbors of a given location.

In the following definitions, we will explain how the new commands work. As a result of the *build* command, the newly built robots will have a tree structure; in terms of this structure:

- *send forward* means send to direct descendants, and
- *send backward* means receive from direct descendants.

In what follows, we will construct "polynomial-time" computing robots, i.e., robots in which the computation time of the corresponding Turing machine grows no more than polynomially with n, and in which the size of the memory also grows no more than polynomially.

Definition 9. *By a* state *of a computing robot, we mean the values of all the bits from its memory. We say that in a given state, a given block of an ID tree memory part is* unused *if it contains only zeros and* used *otherwise.*

Definition 10.

- *By a* robotic computer*, we mean a mapping in which there is a computing robot assigned to (some or all) points from a potentially computation-enhancing space* X*.*
- *By a* state *of the robotic computer, we mean the mapping which assigns to every point, the state of the computing robot assigned to this point.*
- *In the* initial state*, there is only one computational robot located at the user's location* x_0*; its tree ID (i.e., the contents of its tree ID memory) consists of all zeros.*

Comment. The fact in the initial state, the robot's tree ID consists of all zeros means that the value of the "built" bit is 0 (false), and that all the blocks of the tree ID memory are unused.

To complete the description of a robotic computer, we must describe how the new commands work.

Definition 11. *As a result of a "build" command, each robot for which the value of the "built" bit is 0, builds robots in all d_0-neighboring points. These robots are identical to the original robot, with the only exception of an ID number:*

- *All the used blocks of the tree ID of the old robot are copied into the ID tree memory part of the new robot.*
- *In the first previously unused block of the tree ID memory of each new robot, we place 1 in the first bit (to indicate that the new block is used), and assign different values to the next B bits (so that different robots built by the same robot get different tree ID numbers).*

After the new computing robots are built, the original robot changes the value of its "built" bit to 1.

Comments.

- The full description of the state of a robot includes the description of its hardware and its software. Building a new robot means not only building its hardware, but also placing the corresponding program into the newly designed robot. The fact that each new robot is identical to the original robot means that not only their hardware is identical, but also that the new robot also has the same program as the original robot – with the only exception that its ID number is different.
- Every robot has $\le M_0$ neighbors, so it may have to build $\le M_0$ new robots. Due to our choice of B as $B = \lceil \log_2(M_0) \rceil$, B bits are sufficient to distinguish between all these robots.
- If a new point is d_0-close to two or more original robot locations, then several robots want to build a new robot there. To resolve this conflict situation, we can assume, e.g., that a parent robot with the smallest tree number has the priority – i.e., that this smallest-number robot builds a new robot at the new point. It is worth mentioning that our results do not depend on how exactly such conflict situations are resolved. (It is also worth mentioning that there is a relation between this problem and the problem of deleting duplicates in geospatial databases; see, e.g., [43].)

Example. Let us assume that $M_0 = 3$, i.e., that each point has no more than 3 neighbors. In this case, $B = \lceil \log_2(3) \rceil = 2$. So, a tree ID consists of a built bit and blocks of bit size $1 + B = 3$.

Initially, we have only one robot at the user's location x_0 with a tree ID consisting of all zeros: 0 000 000.

When the *build* command is issued, this robot has 0 value of the *built* variable, so it start building new robots. Let us assume that the point x_0 has exactly two neighbors. For each of these neighbors, the first unused block is the first one; its first 0 is replaced by 1, and its next 2 bits are replaced by, e.g., 00 and 11.

So, the two new robots have tree IDs 0 100 000 and 0 111 000. After the initial robot finishes constructing these robots, it changes the value of its *built* bit to 1 ("true"). So, now, its tree ID is 1 000 000.

If the *build* command is issued again, the first robot no longer does anything, because its built variable now has the value 1 ("true"). However, the two new robots have a 0-valued *built* variable, so they both react to this command.

For example, the robot 0 100 000 creates 3 new robots, with tree IDs making the second block used: e.g., 0 100 101, 0 100 110, and 0 100 111. The robot 0 111 000 creates 2 new robots, with tree IDs making the second block used: e.g., 0 111 100 and 0 111 110.

Definition 12. *Robots created by a given robot are called its* direct descendants.

Comment. A robot can easily find its direct parent – by deleting the last used block in its tree ID (and ignoring the built bit). Thus, a robot can easily check whether another robot is its direct descendant.

Definition 13. *As a result of the* send forward *command, each robot sends the values of the memory cells specified by this command. This information is received by all its direct descendants and stored in the first part of their received memory.*

Definition 14. *As a result of the* send backward *command, each robot sends the values of the memory cells specified by this command. This information is received by the direct parent of this robot. The information received by a robot from different direct descendants is stored in the different subparts of this robot's received memory.*

Clarifying comment. We are not specifying where this information is placed, this can be decided by an implementation. For example, each descendant of a given robot can be assigned a number, and then the information sent by the i-th descendant is placed in the i-th subpart of the received memory.

Comment. The above definition describes a SIMD (Single Instruction Multiple Data) type parallel computer in which all the processors, in effect, perform the same set of instructions. In principle, it is possible to extend our definition to the case of MIMD (Multiple Instructions Multiple Data). We did not do this here, since our objective is to describe, in the simplest possible terms, how to solve NP-problems in polynomial time. This can be done already with SIMD-devices only; the description is already somewhat complex, so the extension to MIMD would have made it even more complex and much more difficult to read.

How can we count computation time? In the traditional analysis of the algorithms' computation time, we usually simply count the overall number of elementary computation steps. Of course, this is not a perfect description of the actual computation time, because, in general, different elementary computation steps require somewhat different computation time. For example, multiplication requires slightly longer time than addition, etc. However, these differences are usually minor (less than an order of magnitude), so a simple count of elementary operations provides a reasonable estimation for the actual computation time. (Although, of course, when we need to select the fastest of two reasonable algorithms, counting can be misleading: to make a realistic comparison, we must assign different weights to different operations.)

The situation with computation on a robotic computer is drastically different. Indeed, these computations consists of three types of steps:

- regular computation steps,
- steps in which we build new robots, and
- steps in which the robots send and receive information.

Here, sending a message takes orders of magnitude more time than performing a regular computation step, and building a new robot takes even more time: hours or days instead of nanoseconds.

So, to make a reasonable count of the overall computation time, we cannot simply add the number N_r of the regular computational operations, the number N_b of building operations, and the number N_s of send commands, and take $N \stackrel{\text{def}}{=} N_r + N_b + N_s$ as the computation time. We must define two real numbers $C_b > 0$ ("building time") and $C_s > 0$ ("sending time"), and then estimate the overall computation time as $t = N_r + C_b \cdot N_b + C_s \cdot N_s$.

Intuitively, C_b and C_s are proportional only to the number of robots in the neighborhood (the number of robots to build, or the number of robots to send signals to), so the time needed for each of these longer operations does not change with n; from the viewpoint of dependence on n, it is simply a constant.

The problem is that we do not know these constants. Fortunately, all we are interested in is whether the resulting computation time is polynomial (in terms of th length of the input) or not. Since C_b and C_s are constants, the sum $t = N_r + C_b \cdot N_b + C_s \cdot N_s$ is polynomial if and only if all three components N_r, N_b, and N_s depend polynomially on n – i.e., if and only if the simple sum $N = N_r + N_b + N_s$ polynomially depends on n.

In other words, while the actual computation time depends on the values of the (unknown) coefficients C_b and C_s, the division of algorithms into polynomial-time and not polynomial-time ones does not depend on the actual values of these coefficients. So, we can simply define polynomial time as the case when the sum N is bounded by a polynomial of n.

We are now ready for the main result.

Theorem 1. *For every potentially computation-enhancing space, there exists a robotic computer which solves the propositional satisfiability problem in polynomial time.*

Comment. In our definition, we called a space potentially computation-enhancing if in this space, volume grows exponentially with time – thus providing a potential possibility to fit exponentially many processors within a finite radius and thus, drastically parallelize NP-complete problems into polynomial time.

The above theorem shows that this is not only a *potential* possibility: there is an *actual* algorithmic way to fill such a space with exponentially many computers and thus, indeed enhance computations.

4 Proof of the Main Result

General structure of the algorithm. The desired algorithms consists of the following five stages:

- first, we build a hierarchical network (tree) of 2^n robots;
- second, we assign, to each robot, a number from 0 to $2^n - 1$ (i.e., equivalently, an n-dimensional Boolean vector) in such a way that the numbers assigned to this robot and to the robots eventually generated by this robot and its descendants form an interval in which this robot's number comes first;
- third, we distribute the original propositional formula to all the robots;
- at the forth stage, each robot checks whether the formula F holds at this robot's vector x,
- at the fifth stage, this information is sent backwards along the robot tree until all the information is gathered at the original point – with an indication of whether the original propositional formula is solvable or not.

First (building) stage. Let us first describe the building stage. Initially, we have only one robot in the user's location x_0. One build command creates robots in all the points of distance $\le d_0$ from x_0; another build command creates robots at all the points in locations which are $\le d_0$ far away from these new locations, etc.

Since the space is d_0-connected, every point x is connected to x_0 by a chain in which every two neighboring points are d_0-connected; thus, if we issue one build command after another, eventually, we will cover every point in the space X. How many build commands do we need? For every point x at a distance R, we have a chain of d_0-close points for which $R = d(x, x_0) = d(x_1, x_2) + \ldots + d(x_{m-1}, x_m)$. If in this chain, two points x_i and x_{i+1} are identical, we can delete the second of these two points. Thus, without losing generality,

we can assume that all the points x_i in the chain are different: $x_i \neq x_{i+1}$. Since the space is ε-discrete, $x_i \neq x_{i+1}$ implies that $d(x_i, x_{i+1}) \geq \varepsilon$, hence $R = d(x_1, x_2) + \ldots + d(x_{m-1}, x_m) \geq (m-1) \cdot \varepsilon$. Thus, $m \leq 1 + R/\varepsilon$; this means that every point within the radius R can be covered by $1 + \lfloor R/\varepsilon \rfloor$ iterations of the build command.

What radius R should we choose? The larger the radius, the more robots we will be able to build within this radius. We want to have at least 2^n computing robots. Since we are in a potentially computer-enhancing space, for sufficient large R, the number of points at a distance $\leq R$ from the user's location x_0 is at least $V \geq A \cdot \exp(k \cdot R)$. So, to have $\geq 2^n$ robots, we must have (for sufficiently large n) $\exp(k \cdot R) \geq 2^n/A$, i.e., $k \cdot R \geq n \cdot \ln(2) - \ln(A)$. Thus, it is sufficient to have $R = (\ln(2)/k) \cdot n - (\ln(A)/k)$.

We have already shown that the number of build commands to cover all the points within this radius grows linearly with R as $1 + R/\varepsilon$. Thus, the required number of build commands grows linearly with n, as

$$\frac{\ln(2)}{k \cdot \varepsilon} \cdot n + \left(1 - \frac{\ln(A)}{k \cdot \varepsilon}\right).$$

So, the depth of the resulting tree grows linearly with n. (We need to make sure that the ID tree memory part contains sufficiently many bits to cover that many blocks.)

Second stage: assigning regular IDs to the robots. On the second stage, we assign a number ID to each robot. This is done in two sub-stages. The purpose of the first sub-stage is that each robot computes the number of its descendants (including itself and its indirect descendants). At the end of this sub-stage, the initial robot will have the total number of robots in the whole tree.

The first sub-stage consists of several conditional "send backward" commands:

- the first command is only applicable to the robots of the last generation, with the largest number of used blocks (these robots do not have any direct descendants);
- the next command is applicable only to the robots of the next to last generation,
- etc.,
- finally, the last command is only applicable to the robots of the first generation, with exactly one used block in their tree IDs (i.e., robots built directly by the initial robot).

At each command, each robot sends its tree ID number and its number of descendants to its parent robot. The "leaf" robots (with no descendants) send their tree ID number and the value 1. On every other step, a robot reads all

the values sent by its direct descendants (read from the corresponding parts of the receive part of its memory), adds them together, adds 1, and thus gets the number to send back (to its parent robot).

Each addition of two numbers of size $\le 2^n$ requires n binary steps; thus, addition of $\le M_0$ such numbers requires linear time. Overall, we need $\mathrm{const} \cdot n$ commands each of which requires linear time – to the total of $O(n^2)$ time.

Now, we can distribute regular ID numbers by going forward, from each robot to its direct descendants.This is done by performing a sequence of conditional send forward commands.

- the first command is only applicable to the initial robot;
- the second command is only applicable to the robots of the first generation (directly built by the initial robot);
- etc.
- the last command is applicable only to the robots of the next to last generation.

The initial robot has received values $N_1 + \ldots + N_m$; due to our choice of R, we have $1 + N_1 + \ldots + N_m \ge 2^n$. So, this first robot gets the ID number 0, and the interval $[0, 2^n - 1]$. The first direct descendant gets the interval $[1, N_1]$, the second gets the interval $[N_1 + 1, N_1 + N_2]$, the third gets the interval $[N_1 + N_2 + 1, N_1 + N_2 + N_3]$, etc.

Computing each of these sums requires a constant number of additions; the time of each addition is linear in the number of bits ($\le n$, since each of these sums is $\le 2^n - 1$), so computation of these intervals requires linear time. All these $\le M_0$ intervals and the tree ID number of the direct descendants are then sent to all the direct descendants.

From this information, each direct descendant selects the interval $[\underline{N}, \overline{N}]$ corresponding to its tree ID. Then, this robot assigns the smallest value $\underline{N}$ to itself, and divides the remaining interval $[\underline{N} + 1, \overline{N}]$ between its own direct descendants: if they sent the values $n_1 \ldots, n_s$, then they are assigned the intervals $[\underline{N} + 1, \underline{N} + n_1]$, $[\underline{N} + n_1 + 1, \underline{N} + n_1 + n_2]$, etc.

At the end of these procedure, each robot is assigned an interval of ID values of its descendants, and this robot's own regular ID number is, as we desired, the smallest value from this interval.

Comment. We may have $> 2^n$ robots, in which case values $> 2^n$ are not assigned at all, so the last direct ancestor gets the interval $[\ldots, 2^n]$. This means, in effect, that we will not use all the robots in our computations, only 2^n of them.

Third stage: distributing the propositional formula. On the third stage, the original propositional formula is distributed to all the robots. This is done by applying several "send forward" commands:

- first, from the initial robot to its direct descendants,

- then to the second generation, etc.

We need as many iterations as the height of the tree – i.e., linearly many.

Fourth stage: checking $F(x)$ for every x. On the fourth step no one is sending anything: every robot checks whether $F(x)$ holds for the propositional vector x which is equal to the (binary representation of) its regular ID number.

For example, the initial robot has a regular ID number 0, so it will check whether F holds for the propositional vector $00\ldots0$ consisting of all zeroes (i.e., if all the propositional variables are false). Similarly, the last (childless) robot gets a number 2^n-1 whose binary representation is $11\ldots1$, so it will check whether F holds for the propositional vector $11\ldots1$ consisting of all ones (i.e., if all the propositional variables are true).

Final (fifth) stage: combining the checking results into a single answer to the propositional satisfiability question. Before describing the fifth stage in detail, let us make the following comment. In the original problem, we do not ask for what propositional vector x the formula is satisfied, we just asked whether the formula is satisfied or not. So, in this case, it is sufficient to simply keep the information "yes" (meaning that there is a satisfying vector). In what follows, we show how to modify this problem if we *are* actually interested in finding the vector x for which $F(x)$ holds.

At the beginning of the fifth stage, each robot only has information whether $F(x)$ holds for the vector x corresponding to this robot. There are two possibilities here:

- The first possibility is that $F(x)$ holds for the vector x corresponding to this robot. In this case, we keep the value "yes"; in this situation, the original problem is solved – the formula is satisfiable.
- The second possibility is that $F(x)$ is false for the vector x corresponding to this robot. In this case, we keep the information "no" meaning that $F(x)$ is false for this x.

As we collect this information, at each robot location, we have two similar possibilities:

- The first possibility is that for one of the values x from the corresponding interval $[\underline{x},\overline{x}]$, the formula $F(x)$ is true. In this case, we keep the value "yes". It means that the satisfiability problem is already solved; we only need to make sure that this information is passed back and not lost.
- The second possibility is that $F(x)$ is false for all the vectors x from the interval $[\underline{x},\overline{x}]$ assigned to the node. In this case, we keep the answer "no".

After each robot collects the information from its direct descendants, it has to merge the corresponding $\le M_0$ pieces of information with the information coming from testing $F(x)$ for its own vector x.

- If one of the resulting $\leq M_0 + 1$ pieces of information is "yes", then this "yes" (meaning that there exists a satisfying vector) is the result of the information merger.
- If all of the merged information is "no", this means that $F(x)$ is false for all the vectors x from its interval, so the merged value is "no".

(In other words, this merger is simply an "or" operation.)

Since we combine $\leq M_0 + 1$ bits, this merger requires a constant number of steps $\leq M_0 + 1 = O(1)$.

By the time we get to the original robot – the root of the robot tree – the resulting value "yes" or "no" provides the answer to the original instance of the propositional satisfiability problem.

So, the above-described robotic computer indeed solves the propositional satisfiability problem.

Checking that the resulting algorithm requires polynomial time. To complete the proof, it is now sufficient to show that it requires polynomial time. Indeed:

- Building new computers requires time proportional to the depth of the tree – which, for a potentially computation-enhancing space, grows linearly with the size n.
- Send the information forward and backward to generate regular IDs requires, as we have mentioned, quadratic time ($O(n^2)$).
- Then, checking whether $F(x)$ holds for every x is done on all robots in parallel; it requires polynomial time.
- Finally, merging also requires times which is proportional to the depth of the tree, i.e., linear time.

Thus, the overall time is indeed polynomial.

The theorem is proven.

5 Auxiliary Results

5.1 First auxiliary result: producing satisfying vector instead of checking its existence

We formulated the above algorithm for the *deciding* version of the propositional satisfiability problem, where the objective is simply to check whether a given propositional formula F is satisfiable or not.

One can easily modify this algorithm so that it will serve a similar problem in which we actually want to *produce* the propositional vector x for which $F(x)$

holds. In this case, instead of reporting the single-bit piece of information "yes" to the boss-robot, a robot should pass a vector x for which $F(x)$ holds; if two or more such vectors are passed to a boss-robot, this robot selects one of them (e.g., the smallest in the lexicographic order).

5.2 General problems from the class NP

The known fact that propositional satisfiability is NP-hard means that every problem from the class NP can be reduced, by a polynomial-time reduction, to propositional satisfiability. Thus, we arrive at the following corollary:

Corollary. *For every potentially computation-enhancing space and for every problem from the class NP, an appropriate robotic computer can solve it in polynomial time.*

Proof of the Corollary. Indeed, we can:

- first use the central processor to perform the reduction to satisfiability (this requires polynomial time), and then
- solve the resulting instance of the satisfiability problem in polynomial time (as described in the proof of Theorem 1).

The overall time is thus polynomial. The Corollary is proven.

Comment. In the above description, we make a simplifying assumption: namely, we assume that each computing robot can store (and process) bit strings proportional to the size n of the problem, but we still consider the size of the processor to be fixed (independent of n).

In reality, the linear size ε of the processor should grow with n (e.g., linearly). As a result, when we implement this scheme for a different n, we will need robots separated by n times larger distances. $\varepsilon' \approx n \cdot \varepsilon$. In this case, each communication step requires n times longer time.

Since the product of n and a polynomial of n is still a polynomial of n, this increase keeps the overall time polynomial, so this simplifying assumption does not change the main result.

5.3 Problems from the class PSPACE

Up to now, we have strengthened and generalized the result that in the hyperbolic space, we can solve NP-complete problems in polynomial time. However, in the hyperbolic space, an even stronger result is known. Namely, it is known that by using processors appropriately set up in a hyperbolic space, we can solve, in polynomial time, not only problems from the class NP, but also problems from the more general class PSPACE (the class of all the problems which can be solved within polynomial space).

It turns out that a similar extension can be proved for the case of a general potentially computation-enhancing space.

Theorem 2. *For every potentially computation-enhancing space and for every problem from a class PSPACE, an appropriate robotic computer can solve it in polynomial time.*

Proof of Theorem 2: Main idea. It is known (see, e.g., [41]), that a problem known as QSAT (it will be described later in this proof) is PSPACE-complete. This means that every other problem from the class PSPACE can be reduced to QSAT by a polynomial-time reduction. Thus, to prove that every PSPACE problem can be solved in polynomial time on a robotic computer, it is sufficient to prove that every instance of QSAT can be solved in polynomial time on a robotic computer.

The QSAT problem is an extension of the propositional satisfiability problem SAT, an extension which allows quantifiers over Boolean variables. To be more precise, in SAT, the objective is to find a positional vector $x = (x_1, \ldots, x_n)$ for which a propositional formula $F(x_1, \ldots, x_n)$ becomes true – or, in the deciding version, to check whether such a propositional vector exists, i.e., whether $\exists x\, F(x)$ holds. In QSAT, we must check whether a more general formula

$$\exists x \forall y \ldots \exists z\, F(x, y, \ldots, z)$$

holds.

QSAT is the "limit" of the corresponding formulas from the polynomial hierarchy, which start with SAT-related formulas $\exists x\, F(x)$ and their duals $\forall x\, F(x)$ and continue with more complex formulas $\forall x \exists y\, F(x, y)$ (and $\exists x \forall y\, F(x, y)$), etc., until we reach QSAT.

Let us show, on the example of these next-step formulas $\exists x \forall y\, F(x, y)$, how we can modify the above proof so that it will be applicable to these formulas. The same idea works for all further extensions all the way to QSAT.

In the above proof of Theorem 1, we ordered propositional vectors x in lexicographic order and assigned to each processor an interval $[\underline{x}, \overline{x}]$ of vectors, i.e., all the vectors between the given vectors $\underline{x}$ and $\overline{x}$ ("between" in the sense of the lexicographic order). For the formula $\exists x \forall y\, F(x, y)$, we also sort the processors in lexicographic order by the order of a "long vector" xy (a concatenation $x_1, \ldots, x_n, y_1, \ldots, y_m$ of the vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$) and assign, to each processor, all the vectors (x, y) from some interval (interval in the sense of this order).

The lexicographic ordering has the property that if the vectors $x0 \ldots 0$ and $x1 \ldots 1$ belong to an interval, then all the vectors xy belong to the same interval. Thus, in general, the interval in this order can be described as follows:

- for the smallest possible x from this interval, we have all the vectors xy corresponding to a subinterval of possible values of y;

- for some (maybe empty) interval $[\underline{x}, \overline{x}]$ of possible values of x, we have all the values xy with $x \in [\underline{x}, \overline{x}]$ and arbitrary y;
- finally, for the last x, we have all the vectors xy corresponding to a subinterval of possible values of y.

(In the degenerate case, we may have no intermediate interval, or no starting interval, or no ending interval.)

By the time we gathered information from this xy-interval at a robotic computer, we have already checked whether $F(x, y)$ holds for all (x, y) within this interval, and have already collected this information.

For the values x from the intermediate x-interval $[\underline{x}, \overline{x}]$, we thus already know whether $F(x, y)$ holds for all y. If for one of these x, it is true that $\forall y\, F(x, y)$, then the original problem (of checking whether $\exists x \forall y\, F(x, y)$ is true) is solved: we just need to keep this information and make sure that it is not lost during the transition to the boss-robots.

If no y was found for all these x, then we have an interval $[\underline{x}, \overline{x}]$ of values for which $\neg\forall y F(x, y)$. So, we simply keep the endpoints of this interval.

Similarly, at each of the two borderline values x, if there is a y (among those for which xy was checked) for which $\neg F(x, y)$, then the resulting x can be simply added to the intervals of xs for which $\neg\forall y F(x, y)$. Otherwise, for each of these two borderline values x, we keep an interval $[\underline{y}, \overline{y}]$ for which $F(x, y)$ holds for all y within this interval.

As a result, at each robot location, we keep one of the two following pieces of information:

- first possibility is the information that the problem is already solved;
- second possibility is that we keep
  - the interval $[\underline{x}, \overline{x}]$ of values x for which $\neg\forall y F(x, y)$, and
  - for two borderline values x, interval of values $[\underline{y}, \overline{y}]$ for which $F(x, y)$ holds for all y within these intervals.

At each robot point, we combine several ($\leq M_0{+}1$) such pieces of information into one. If one of these pieces contains the solution, then we just pass this information on without bothering with other piece of information. If none of these pieces contain a solution, then first, for some borderline values x, we may need to combine the y-intervals.

- It may be possible that we simply get a larger interval of possible values of y.
- As a result of the combination, we may conclude that $F(x, y)$ holds for all possible y – in which case we have a solution.

Since we combine $\leq M_0 + 1$ pieces of information, there are $\leq M_0$ borderline values to combine, so this merger can done in constant time.

This is the only modification that we need in the above proof to convert from SAT to $\exists x \forall y\, F(x, y)$.

For more complex formulas such as $\exists x \forall y \exists z\, F(x, y, z)$, an interval in the lexicographic order includes:

- an interval of possible values of x for which we have checked for all y and z,
- two borderline values (x, y) for which we have checked for intervals of z, etc.

The theorem is proven.

Acknowledgments. This work was supported in part by NSF grants HRD-0734825 and EAR-0225670, by Texas Department of Transportation grant No. 0-5453, and by the Japan Advanced Institute of Science and Technology (JAIST) International Joint Research Grant 2006-08. This work was partly done during Maurice Margenstern's visit to El Paso, Texas and Vladik Kreinovich's visit to the Max Planck Institut für Mathematik.

The authors are very thankful to Roberto Araiza for his help and to the anonymous referees for valuable suggestions.

References

[1] L. M. Adleman, "Molecular Computation Of Solutions To Combinatorial Problems", *Science*, 1994, Vol. 266, No. 11, pp. 1021–1024.

[2] M. Amos, *Theoretical and Experimental DNA Computation*, Springer Verlag, 2005.

[3] M. Arbib, "Simple self-reproducing universal automata", *Inf. Control*, 1966, Vol. 9, pp. 177–189.

[4] M. A. Arbib, *Theories of Abstract Automata*, Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

[5] M. A. Arbib, "The Likelihood of the Evolution of Communicating Intelligences on Other Planets", In: C. Ponnamperuma and A. G. W. Cameron, *Interstellar Communication: Scientific Perspectives*, Houghton Mifflin Co., Boston, Massachushetts, 1974, pp. 63–66.

[6] A. Blass and Yu. Gurevich, "Algorithms: A Quest for Absolute Definitions", *Bulletin of European Association for Theoretical Computer Science*, 2003, Vol. 81, pp. 195–225.

[7] R. Bonola, *Non-Euclidean Geometry*, Dover Publications, Inc., New York, 1955.

[8] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, Berlin-Heidelberg-New York, 2003.

[9] C. Boyce, *Extraterrestrial Encounter: A Personal Perspective*, David & Charles, Newton Abbot, London, 1979.

[10] K. Chelghoum, M. Margenstern, B. Martin, and I. Pecci, "Cellular Automata in the Hyperbolic Plane: Proposal for a New Environment", In: P. M. A. Sloot, B. Chopard, A. G. Hoekstra (Eds.), *Cellular Automata, Proceedings of the 6th International Conference on Cellular Automata for Research and Industry ACRI'2004*, Amsterdam, October 25–28, 2004, Springer Lecture Notes in Computer Science, 2004, Vol. 3305, pp. 678–687.

[11] S. Dasgupta, C. Papadimitriou, and U. Vazirani, *Algorithms*, McGraw-Hill, Boston, Massachusetts, 2006.

[12] Y. Feldman and E. Shapiro, "Spatial machines: a more realistic approach to parallel computations", *Communications of the ACM*, 1992, Vol. 35, No. 10, pp. 61–73.

[13] R. A. Freitas Jr., "A Self-Reproducing Interstellar Probe," *Journal of the British Interplanetary Society*, 1980, Vol. 33, pp. 251–264.

[14] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. F. Freeman, San Francisco, 1979.

[15] S. Grigorieff and M. Margenstern, "Register Cellular Automata in the Hyperbolic Plane", *Fundamenta Informaticae*, 2004, Vol. 61, No. 1, pp. 19–27.

[16] D. Grigoriev, "Kolmogorov algorithms are stronger than Turing machines", *Zapiski Nauchnykh Seminarov LOMI* (*Notes of Mathematical Seminars of Leningrad Department of V. A. Steklov Institute of Mathematics*), 1976, Vol. 60, pp. 29–37 (in Russian); English translation in *Journal of Soviet Mathematics*, 1980, Vol. 14, No. 5, pp. 1445–1450.

[17] Yu. Gurevich, "On Kolmogorov Machines And Related Issues", *Bulletin of European Assoc. for Theor. Comp. Science*, 1988, Vol. 35, pp. 71–82.

[18] Yu. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", *Bulletin of the European Association for Theoretical Computer Science,* 1991, Vol. 43, pp. 264–284).

[19] Yu. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", In: G. Rozenberg and A. Salomaa (eds.), *Current Trends in Theoretical Computer Science*, World Scientific, Singapore, 1993, pp. 266–292.

[20] Yu. Gurevich, "Evolving Algebras 1993: Lipari Guide", In: E. Borger (ed.), *Specification and Validation Methods*, Oxford University Press, Oxford, UK, 1995, pp. 9–36.

[21] Yu. Gurevich, "Sequential abstract-state machines capture sequential algorithms", *ACM Transactions on Computational Logic (TOCL)*, 2000, Vol. 1, No. 1, pp. 77–111.

[22] F. Herrmann and M. Margenstern, "A universal cellular automaton in the hyperbolic plane", *Theor. Comput. Sci.*, 2003, Vol. 296, No. 2, pp. 327–364.

[23] A. N. Kolmogorov, "On the concept of algorithm", *Uspekhi Mat. Nauk*, 1953, Vol. 8, No. 4, pp. 175–176 (in Russian); an English translation is found in [45], pp. 18–19.

[24] A. N. Kolmogorov and V. A. Uspensky, "On the definition of algorithm", *Uspekhi Mat. Nauk*, 1958, Vol. 13, No. 4, pp. 3–28 (in Russian); English translation in *American Math. Soc. Translations*, 1963, Vol. 29, pp. 217–245.

[25] V. Kreinovich and G. Mints (eds.), *Problems of reducing the exhaustive search*, American Mathematical Society, Providence, RI, 1997.

[26] L. A. Levin, "Universal Search Problems", *Problemy Peredachi Informatsii*, 1973, Vol. 9, No. 3, pp. 265–266 (in Russian); the journal is translated into English as *Problems of Information Transmission*; English translation is also presented in [44].

[27] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Inc., New Jersey, 1997.

[28] M. Margenstern, "New Tools for Cellular Automata in the Hyperbolic Plane", *Journal of Universal Computer Science*, 2000, Vol. 6, No. 12, pp. 1226–1252.

[29] M. Margenstern, "A Combinatorial Approach to Hyperbolic Geometry as a New Perspective for Computer Science and Technology", In: N. C. Debnath (Ed.), *Proceedings of the 18th International Conference on Computers and Their Applications ISCA'03*, Honolulu, Hawaii, USA, March 26–28, 2003, pp. 468–471.

[30] M. Margenstern, "Cellular Automata and Combinatoric Tilings in Hyperbolic Spaces. A Survey", In: C. Calude, M. J. Dinneen, V. Vajnovszki (Eds.), *Proceeding of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science DMTCS'03*, Dijon, France, July 7–12, 2003, Springer Lecture Notes in Computer Science, 2003, Vol. 2731, pp. 48–72.

[31] M. Margenstern, "The Tiling of the Hyperbolic 4D Space by the 120-cell is Combinatoric", *Journal of Universal Computer Science*, 2004, Vol. 10. No. 9, pp. 1212–1238.

[32] M. Margenstern, "A new way to implement cellular automata on the penta- and heptagrids", *Journal of Cellular Automata*, 2006, Vol. 1, No. 1, pp. 1–24.

[33] M. Margenstern, "An Algorithm for Building Intrinsically Universal Automata in Hyperbolic Spaces", In: *Proceedings of the 2006 International Conference on Foundations of Computer Science*, Las Vegas, Nevada, USA, June 26–29, 2006, pp. 3–9.

[34] M. Margenstern, *Cellular Automata in Hyperbolic Spaces*, Old City Publishing, Philadelphia, Pennsylvania, Vol. 1, 2007; Vol. 2, to appear.

[35] M. Margenstern and K. Morita, "A Polynomial Solution for 3-SAT in the Space of Cellular Automata in the Hyperbolic Plane", *Journal of Universal Computer Science*, 1999, Vol. 5, No. 9, pp. 563–573.

[36] M. Margenstern and K. Morita, "NP problems are tractable in the space of cellular automata in the hyperbolic plane", *Theor. Comput. Sci.*, 2001, Vol. 259, No. 1–2, pp. 99–128.

[37] M. Margenstern and G. Skordev, "Tools for devising cellular automata in the hyperbolic 3D space", *Fundamenta Informaticae*, 2003, Vol. 58, pp. 369–398.

[38] S. Yu. Maslov, *Theory of Deductive Systems and Its Applications*, MIT Press, Cambridge, Massachusetts, 1987.

[39] Yu. Matiyasevich, "Possible nontraditional methods of establishing unsatisfiability of propositional formulas", In: V. Kreinovich and G. Mints (eds.), *Problems of reducing the exhaustive search*, American Mathematical Society (AMS Translations - Series 2, Vol. 178), Providence, RI, 1997, pp. 75–77 (originally published in Russian in 1986).

[40] 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.

[41] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[42] W. Reisig, "On Gurevich's Theorem on Sequential Algorithms", *Acta Informatica*, 2003, Vol. 39, No. 5, pp. 273–305.

[43] R. Torres, G. R. Keller, V. Kreinovich, L. Longpré, and S. A. Starks, "Eliminating Duplicates Under Interval and Fuzzy Uncertainty: An Asymptotically Optimal Algorithm and Its Geospatial Applications", *Reliable Computing*, 2004, Vol. 10, No. 5, pp. 401–422.

[44] B. A. Trakhtenbrot, "A Survey of Russian Approaches to Perebor (Brute-force Search) Algorithms", *Annals of the History of Computing*, 1984, Vol. 6, No. 4, pp. 384–400.

[45] V. A. Uspensky and A. L. Semenov, *Algorithms: Main Ideas and Applications*, Kluwer, Dordrecht, 1993.

[46] F. Valdes and R. A. Freitas, "Comparison of Reproducing and Non-Reproducing Starprobe Strategies for Galactic Exploration", *Journal of the British Interplanetary Society*, 1980, Vol. 33, pp. 402–408.

[47] J. von Neumann and A. W. Burks, *Theory of self-reproducing automata*, University of Illinois Press, Urbana, Illinois, 1966.

[48] J. von Neumann, "Theory of Self-Reproducing Automata", In: A. W. Burks (ed.), *Essays on Cellular Automata*, University of Illinois, Urbana, Illinois, 1969, pp. 4–65.

[49] G. von Tiesenhausen and W. A. Darbro, *Self-Replicating Systems*, NASA Technical Memorandum 78304, National Aeronautics and Space Administration (NASA), Washington, D.C., 1980.

[50] V. Zykov, E. Mytilinaios, B. Adams, H. Lipson, "Self-reproducing machines", *Nature*, 2005, Vol. 435, No. 7038, pp. 163–164.