8-1-2011

High-Concentration Chemical Computing Techniques for Solving Hard-To-Solve Problems, and Their Relation to Numerical Optimization, Neural Computing, Reasoning Under Uncertainty, and Freedom Of Choice

Vladik Kreinovich
University of Texas at El Paso, vladik@utep.edu

Olac Fuentes
University of Texas at El Paso, ofuentes@utep.edu

Comments:
Technical Report: UTEP-CS-11-44
To appear in: Evgeny Katz (ed.), *Molecular and Biomolecular Information Processing*, Wiley-VCH.

High-Concentration Chemical Computing Techniques for Solving Hard-To-Solve Problems, and Their Relation to Numerical Optimization, Neural Computing, Reasoning Under Uncertainty, and Freedom Of Choice

Vladik Kreinovich and Olac Fuentes
Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
{vladik, ofuentes}@utep.edu

Abstract

Chemical computing – using chemical reactions to perform computations – is a promising way to solve computationally intensive problems. Chemical computing is promising because it has the potential of using up to 10^{23} molecules as processors working in parallel – and thus, has a potential of an enormous speedup. Unfortunately, for hard-to-solve (NP-complete) problems a natural chemical computing approach for solving them is exponentially slow. In this chapter, we show that the corresponding computations can become significantly faster if we use very-high-concentration chemical reactions, concentrations at which the usual equations of chemical kinetics no longer apply. We also show that the resulting method is related to numerical optimization, neural computing, reasoning under uncertainty, and freedom of choice.

1 What Are Hard-so-Solve Problems and Why Solving Even One of Them Is Important

What is so good about being able to solve hard-to-solve problems from some exotic class? In this paper, we will talk about applying chemical computing to a specific class of hard-to-solve (NP-complete) problems.

To a person who is not very familiar with the notions of NP-completeness, this may sound like a very exotic (and thus not very interesting) topic. For example, this person may ask: OK, we spend all these efforts and solve problems from this exotic class, how will this help me solve my own hard-to-solve problems, problems which are formulated in completely different terms?

- A *short answer* to this equation is: once we learn how to solve problems from *one* class, then we will be able to solve *all* hard-to-solve problems.
- A *detailed answer* to this question – with appropriate explanations – is given in this section.

Since this volume is devoted to chemical computing, we expect most readers to be familiar with at least the basics of chemistry. Because of this, in our description of NP-completeness, we will try (whenever possible) to use examples from (computational) chemistry. Since some potential readers are computational scientists, who may not be very familiar with the details of computational chemistry problems, we will try to explain the related chemistry problems as much as possible.

Comment. Readers who are already very well familiar with the notions of P, NP, and NP-completeness are welcome to skip this section. Readers interested in more details can read, e.g., [9, 14].

In many applications areas – in particular, in chemistry – there are many well-defined complex problems. In many application areas, we face well-defined problems. Many such problems are known in chemistry.

For example, it is known that from the physical viewpoint, chemical reactions are interactions between electrons of different atoms. Thus, to get a good understanding of the chemical reactions, it is desirable to describe possible electronic states and their energies. There exist known fundamental equations – partial differential equations originally proposed by Schrödinger – that exactly describe these states.

On a more large-scale level, changes in concentrations that occur during a complex chemical reaction are described by a system of ordinary differential equations – equations of chemical kinetics. If we also want to take into account spatial inhomogeneities, we need to use the corresponding partial differential equations, etc.

In some applications, we need to solve optimization problems. For example, in bioinformatics applications, we know how to describe, for each possible folding of a protein, the resulting potential energy. Based on this description, we need to find the folding for which this potential energy is the smallest possible – because this is the shape into which proteins fold within a cell.

In principle, there exist algorithms for solving these problems. In computational mathematics, there exist algorithms for solving the corresponding problems: i.e., algorithms for solving systems of ordinary or partial differential equations, algorithms for finding where a complex function attains its minimum, etc.

These algorithms may take too much time to be practical. The problem is that often, these algorithms, when applied to practical chemistry-related (and other) problems, require too much time to be practically useful.

Feasible and unfeasible algorithms: general idea. When the algorithm takes too much time, the big question is how much. For some problems, the required computation time is, e.g., ten or hundred times larger that what we have accessible now. In this case, there is a good chance that this problem will be soon solved:

- it can either be solved right away, by running the algorithm on a high-performance supercomputer – which is usually several orders of magnitude faster than usual university computers,
- or it can be solved in a few years, since the computer speed approximately doubles every few years or so (this empirical fact is known in Computer Science as *Moore's law*).

Informally, we can say that such algorithms may not be practical on a typical computer, they may not be practical on all existing computers – but they are *feasible* in the sense that in reasonable amount of time, and with appropriate resources, these algorithms can be implemented.

On the other hand, there are other algorithms for which required computation time may be 10^{20} times larger than what we have available; an example will be given soon. For such an algorithms, we can use the fastest supercomputers, we can wait 10 years – none of this will overcome this enormous gap between the desired computation speed and the available speed of computations. From the practical viewpoint, such algorithms are *unfeasible.*

Let us give examples of feasible and unfeasible algorithms.

Solving equations of chemical kinetics: an example of a feasible algorithm. Let us consider the most realistic case of equations that take into account the spatial inhomogeneity. These partial differential equations describe the dependence $c_i(x, y, z, t)$ of the concentration of each substance i at different spatial points (described by spatial coordinates x, y, and z) at different moments of time.

Most computational techniques for solving partial differential equations are based on the following straightforward idea:

- instead of considering all infinitely many moments of time, we consider only moments on a grid, e.g., moments t_0, $t_1 = t_0 + \Delta t$, $t_2 = t_1 + \Delta t = t_0 + 2\Delta t$, $\ldots$, $t_k = t_0 + k \cdot \Delta t$, $\ldots$, for some small step Δt;
- similarly, instead of considering all infinitely many values (x, y, z), we consider finitely many points on a grid – e.g., values with $x = x_0 + k_x \cdot \Delta x$, $y = y_0 + k_y \cdot \Delta y$, and $z = z_0 + k_z \cdot \Delta z$.

Of course, this is an over-simplified description. In more sophisticated algorithms, such as Finite Element methods, instead of using a pre-determined grid, we select points as we go – more points in areas of drastic change and fewer points in areas where there is practically no spatial dependence.

In this discrete approximation, the original partial differential equation becomes a difference equation, and we can solve it by consequently computing the values at moment t_0, at moment t_1, etc. If N is a total number of grid points in each of the four directions x, y, z, and t, then we have N^4 possible combinations of these values, i.e., N^4 nodes on a 4-D grid at each of which we need to perform appropriate computations.

If we have n substances i, and we only consider reactions with two inputs (i.e., of the type $i + j \to \ldots$), the the right-hand side of the corresponding equations of chemical kinetics contains terms like $c_i \cdot c_j$. Since we have n different substances, there are no more than n^2 such terms, and thus, the total computation time grows with number of substances as n^2.

In this case, if we double the number of substances – e.g., take into consideration important short-term intermediate substances whose study is very important for studying catalysis – the whole computation time increases only by a factor of four. We can usually afford such an increase – either by waiting four times longer for the computations to finish, or by going to a need to a four times faster computer, or by waiting $2 + 2 = 4$ years during which, according to Moore's law, computers will twice double in speed and thus, become $2 \times 2 = 4$ times faster.

If we take into account reactions with 3 inputs, then the computation time starts growing as n^3. If we double n, the total computation time increase by a factor of eight – still feasible.

So, straightforward algorithms for solving equations of chemical kinetics are feasible.

Straightforward solution of Schrödinger equation: an example of an unfeasible algorithm. Schrödinger's equation is the main equation of quantum physics. To describe an atom with n electrons in quantum physics, we need to describe a complex-valued function $\Psi(t, x_1, y_1, z_1, \ldots, x_n, y_n, z_n)$ called *wave function*. Here, x_i, y_i, and z_i are spatial coordinates of the i-th particle. A straightforward way to solve this partial differential equation is the same as for chemical kinetics: select a grid and consider only points from a grid. The difference is that when we select N options for x_1, N options for y_1, N options for z_1, $\ldots$, and N options for z_n, then we get N^{3n+1} possible combinations (grid points) $(t, x_1, y_1, z_1, \ldots, x_n, y_n, z_n)$. Processing each grid point requires at least one computational step, so the overall number of computational steps – and thus, the overall computation time – grows exponentially with n, as c^n for some constant c.

Such computations are realistically possible for the hydrogen H for which there is one electron ($n = 1$), possible for the helium He for which $n = 2$, but, e.g., for the iron Fe, with $n = 26$, even for the simplest case when we take only two points $N = 2$ in each direction, we need $2^{3n+1} = 2^{79} \approx 3 \cdot 10^{24}$ steps. The fastest supercomputer performs 10^{12} operations per second, so in a year, it can perform $3 \cdot 10^7$ sec/year $\cdot 10^{12}$ oper/sec $= 3 \cdot 10^{19}$. Thus, we need 30 000 years to finish these computations – and we only considered a rather useless approximation with only two values per spatial dimension. For a somewhat better (but still lousy) approximation, we can take $N = 10$ points per dimension, in this case we need 10^{79} steps: much more than the fastest computer can perform during the lifetime of the Universe.

This algorithm is clearly unfeasible.

Straightforward approach to protein folding: another example of an unfeasible algorithm. A guaranteed way to find a global minimum of a function of n variables is to compute its values on all the points of a grid and to find the smallest of these values. This methods requires N^n steps and is, thus, feasible, e.g., for $n = 1$, $n = 2$, even $n = 3$ variables. However, in the protein folding, we need to find spatial locations of several thousand atoms forming the protein, so $n \approx 10^3$, and the value N^n is not even astronomical: it is much much larger that the lifetime of the Universe.

Feasible and unfeasible algorithms: towards a formal description. In the above examples, for some algorithms, the computation time grows polynomially with the number n of inputs, as $C \cdot n^k$ for some k; these algorithms were feasible. For some algorithms, the computation time grows exponentially with n, as c^n; these algorithms were unfeasible.

This distinction underlies the current formal definition of a feasible algorithm: an algorithm is called *feasible* if there exists a polynomial P such that on every input of size n, this algorithm finishes computations in time $\le P(n)$. All other algorithms are considered *unfeasible.*

Comment. It is well known that this definition does not always properly capture the intuitive idea of feasibility. For example, an algorithm that requires computation time $10^{40} \cdot n$ is not practically feasible but it is feasible in the sense of the above definition. On the other hand, an algorithm that requires time $\exp(10^{-9} \cdot n)$ is practically feasible but not feasible in the sense of the above definition – since the exponential function cannot be bounded from above by any polynomial. However, this is the best definition we have :-(

Maybe the problem itself is hard-to-solve? When an algorithm for solving a problem is not feasible, a natural idea is to look for a faster algorithm. But maybe it is not the algorithm's fault? maybe the problem itself is hard to solve, so that no feasible algorithm is possible that would solve all particular cases of this problem?

To be able to decide whether a problem is hard to solve, we need to first provide precise definition of what is a problem and what does it mean for a problem to be hard to solve. Let us start with describing what is a problem.

What is a problem in the first place? In the previous discussion, we tried our best to relate to chemistry. However, when we analyze what is a problem, we want our answer to be as general as possible – to make sure that we do not miss important real-life problems. Thus, let us now consider the activity of other disciplines as well.

What is a problem: mathematics. For example, the main activity of a mathematician is proving theorems. We are given a mathematical statement x, and we need to find a proof y:

- either a proof that x is true
- or, if x is not true, a proof that the original statement x is false.

Mathematicians are usually interested in proofs which can be checked by human researchers, and are, thus, of reasonable size. This notion of "reasonable size" can be formalized in the same way as in the definition of a feasible algorithm: as the existence of a polynomial P_l for which the length $\text{len}(y)$ of the proof y does not exceed the result $P_l(\text{len}(x))$ of applying this polynomial to the length $\text{len}(x)$ of the input x.

In the usual formal systems of mathematics, the correctness of a formal proof can be checked in polynomial time. So, the main problem of mathematics can be formulated as follows:

A description of a general problem.

- *A description of a general problem.* We are given:
  - a feasible algorithm $C(x, y)$ that, given two strings x and y, returns "true" or "false"; and
  - a polynomial P_l.
- *A description of the particular case (instance) of the general problem.*
  - we are given a string x;
  - we must find a string y of length $\text{len}(y) \leq P_l(\text{len}(x))$ for which $C(x, y) =$"true" – or produce the corresponding message if there is no such string.

Comment. The possibility that we have neither a proof of x nor a proof of its negation $\overline{x}$ is quite real: there are known statements x which are independent of the axioms.

What about other activity areas? The above description was derived from the analysis of mathematics, but, as we will now show, a similar description applies to other activity areas as well.

What is a problem: theoretical physics. In theoretical physics, one of the main challenges is to find a formula y that describe the observed data x. The size of such a formula cannot exceed the amount of data: otherwise, we could simply enumerate all the observations and call it a formula. So, here, $\text{len}(y) \leq \text{len}(x)$, i.e., we have the above inequality for $P_l(n) = n$. Once a formula y is proposed, it is easy to check whether it is consistent with all the observations x: this can be done observation-by-observation, so this checking can be performed in linear time. If we denote by $C(x, y)$ the statement "the formula y is consistent with observations x", then we get exactly the above formulation.

What is a problem: engineering. In engineering, one of the main challenges is to find a design y that satisfies given specifications x. For example, a design for a bridge must be able to withstand winds up to a certain speed and loads up to a certain amount, and its building cost should not increase the amount allocated in the budget.

This design has to be practical, so its description cannot be too long; thus, a condition of the type $\text{len}(y) \leq P_l(\text{len}(x))$ sounds quite reasonable. Once a design y is proposed, we can use known engineering software tools to efficiently check whether the design y satisfies the specifications x; so, we have a feasible checking algorithm $C(x, y)$. Thus, we also get exactly the above formulation.

Class NP. In all these general problems, once we guess a solution candidate y, we can check, in polynomial time, whether this guess y is indeed a solution. In theoretical computer science, computations with guessing steps are called *nondeterministic*. Thus, this class is called Nondeterministic Polynomial, or, for short, NP.

Class P and the P$\overset{?}{=}$NP problem. For some of the problems from the class NP, there exist algorithms which solve these problems in polynomial time (i.e., feasibly). The class of all such problem is denoted by P.

By definition, the class P is a subset of the class NP: P$\subseteq$NP. A natural question is: is P a proper subclass of NP? In other words, do there exist problems from the class NP that cannot be solved in polynomial time – or, vice versa, every problem from the class NP can be feasibly solved and thus, P=NP? The answer to this question is unknown. Checking whether P is equal to NP is an open problem for 40 years already. Most computer scientists believe that these classes are different, but no one knows for sure.

Exhaustive search: why it is possible and why it is not feasible. In principle, since the length $\text{len}(y)$ of a possible solution is a priori restricted (by the value $P_l(\text{len}(x))$), we can simply try all the words y of length $\leq P_l(\text{len}(x))$ until we find a string y that satisfies the desired condition $C(x, y)$. There are finitely many words of given length, so this procedure always produces the desired result.

This "exhaustive search" algorithm works for small lengths, but in general, this algorithm is not feasible. Indeed, even for the binary alphabet, we need to try $2^{P_l(\mathrm{len}(x))}$ possible words y, and we have already shown that even for reasonable values m, it is not feasible to perform 2^m computational steps.

Notion of NP-complete problems. The fact that no one knows whether P is equal to NP does not mean that we have no information about the relative complexity of different problems from the class NP. There are known *reducibility* relations between different problems A and A': sometimes, every instance of the problem A can be feasibly reduced to an instance of the problem A'. In this case, the problem A' is harder than – of the same hardness as – the problem A, in the sense that if we can efficiently solve every instance of the problem A', then we can also solve every instance of the problem A.

For example, if we know how to solve systems with three unknowns, then we can solve every system with two unknown – by introducing a dummy third variable and applying the algorithm for solving systems with three unknowns. Thus, solving systems with three unknowns is harder than (or of the same hardness as) solving systems of two unknowns.

Similarly, if we know how to solve a system of linear *inequalities*, then we can also solve systems of linear *equalities* – since each equality $f = 0$ is equivalent to two inequalities $f \geq 0$ and $f \leq 0$. Thus, solving systems of linear inequalities is harder than (or of the same hardness as) solving systems of linear equalities.

An important discovery made in the early 1970s – a discovery that started the whole area of research about P, NP, and NP-completeness – that in the class NP, there exist problems to which every other problem from the class NP can be reduced. Thus, each of these problems is harder than (or of the same quality as) the complete class NP. Such hard-to-solve problems are called *NP-complete.*

Why solving even one NP-complete (hard-to-solve) problem is very important. Because of the reduction-related definition of NP-completeness, once we know how to efficiently solve *one* NP-complete problem, we will then be able to efficiently solve *all* problems from the class NP. Similarly, once we have an algorithm that efficiently solves *many* instances of one NP-complete problem, we can the reduction to solve many instances of other problems from the class NP.

Thus, any progress in solving *one* of NP-complete problems automatically leads to a progress in *all* of them. As a result, solving even one NP-complete problem — no matter how exotic is looks, no matter how unrelated it seems to the problems in which we are actually interested – is very important because it will help other problems.

Propositional satisfiability: historically the first NP-complete problem. At present, thousands of different NP-complete problems are known. Historically, the first problem for which NP-completeness was proved was the *propositional satisfiability* problem. This problem is still actively studied as an example of an NP-complete problem, so let is describe this problem.

For convenience, instead of describing the original satisfiability problem, we will describe an easy-to-describe class 3-SAT which is also known to be NP-hard. We start with n propositional variables $v_1, \ldots, v_n$, i.e., variables each of which can take only two values: "true" (usually represented, in the computers, by 1) and "false" (usually represented, in the computers, by 0). By a *literal* a, we mean a variable v_i or its negation $\overline{v}_i$. By a *clause* C, we means an expression of one of the following types: $a \vee b$ or $a \vee b \vee c$, where a, b, and c are literals. Finally, by a *formula* F, we mean an expression of the type $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$, where $C_1, \ldots, C_m$ are clauses.

To illustrate this concept, let us give a simple example of the formula:

$$(v_1 \vee \overline{v}_2) \,\&\, (v_1 \vee v_2 \vee v_3).$$

This formula had $m = 2$ clauses: $v_1 \vee \overline{v}_2$ and $v_1 \vee v_2 \vee v_3$.

The problem is: *given* a formula F, *find* the values of the variables $v_1, \ldots, v_n$ that make this formula true (i.e., for which the formula F is *satisfied*) – or return a message that such values do not exist. Once we have a sequence of values $v_1, \ldots, v_n$, we can plug these values into the formula F and easily check whether the formula is true. Thus, this problem belongs to the class NP. (The proof that this problem is NP-complete is beyond the scope of this chapter.)

What we do. In this chapter, we study the above-described propositional satisfiability problem: namely, we show how chemical computing can solve this problem.

2 How Chemical Computing Can Solve a Hard-to-Solve Problem of Propositional Satisfiability

Chemical computing: main idea. When a person needs to perform a complex task – e.g., build a house, dig a ditch – and realizes that it would take too much time for him to do it alone, he gets himself a helper. When they work simultaneously, in parallel, they finish the task faster. To perform this task even faster, we can get many helpers, the more helpers (up to a certain limit), the better.

Similarly, when a computational problem requires too much computation time, a natural way to finish computations faster is to have many computers working in parallel. From this viewpoint, what can be faster than having all $\approx 10^{23}$ molecules work in parallel to perform the desired computations? In other works, ideally, we should make chemical reactions – on the level of individual molecules – perform the desired computations.

This is the main idea behind chemical computing.

Why propositional satisfiability was historically the first problem for which a chemical computing scheme was proposed. This may be not a widely known fact, but the main idea of chemical computing was first proposed by Yuri Matiyasevich exactly for the purpose of solving a hard-to-solve problem of propositional satisfiability. This idea was first presented at a meeting; it was first published in [13].

It makes sense to have selected a hard-to-solve problem: these problem require a lot of computations time, and thus, for them, the need to reduce this time is the most ungent. But why propositional satisfiability and not any other hard-to-solve problem? The answer is that, surprisingly, the propositional satisfiability problem can be naturally represented in terms which are very similar to chemistry.

To explain this representation, let us recall the meaning of each clause. A clause $a \vee b$ means that either a is true or b is true. Thus, if a is false, then b is true; similarly, if b is false, then a should be true. In other words, this clause can be represented by two implications

$$\overline{a} \to b; \quad \overline{b} \to a.$$

Vice versa, if both these implications are true, this means that the clause $a \vee b$ is true. Indeed, in general, either a is true or a is false.

- If a is true, then the clause $a \vee b$ is also true.
- If a is false, then, due to the implication $\overline{a} \to b$, the literal b is true.

In both cases, the clause is true. (Notice that we used only one implication.)

Similarly, a clause $a \vee b \vee c$ means that one of the three literals a, b, and c must be true. Thus, if both a and b are false, then c must be true; if a and c are both false, then b must be true; and if b and c are both false, then a must be true. In other words, this clause can be represented by three implications:

$$\overline{a}, \overline{b} \to c; \quad \overline{a}, \overline{c} \to b; \quad \overline{b}, \overline{c} \to a.$$

Vice versa, one can check that if these implications are true, then the original clause is true is well. (Actually, it is sufficient to require that one of these implications is true.)

For example, the above formula $(v_1 \vee \overline{v}_2) \,\&\, (v_1 \vee v_2 \vee v_3)$ can be represented by the following five implications:

$$\overline{v}_1 \to \overline{v}_2; \quad v_2 \to v_1; \quad \overline{v}_1, \overline{v}_2 \to v_3; \quad \overline{v}_1, \overline{v}_3 \to v_2; \quad \overline{v}_2, \overline{v}_3 \to v_1.$$

How to apply chemical computing to propositional satisfiability: Matiyasevich's original idea. Matiyasevich noticed that these implications look exactly like chemical reactions involving substances v_i and $\overline{v}_i$. Thus, he proposed to solve the original propositional satisfiability problem with variables $v_1, \ldots, v_n$ by finding $2n$ substances which have exactly these implications $a, b \to c$ as chemical reactions $a + b \to c$. For each variable v_i:

- the larger concentration of the substance v_i in comparison with the concentration of the "opposite" substance $\overline{v}_i$ indicates that this variable v_i is true, while

- the larger concentration of substance $\overline{v}_i$ in comparison with the concentration of v_i indicates that v_i is false.

These reactions work in such a way as to make all the implications true, and thus, the whole formula true. For example, the reaction $\overline{a} \to b$ means that if we have a prevalence of the substance $\overline{a}$ (i.e., if, in our interpretation, a is false), then this reaction would create a prevalence of the substance b – i.e., b will become true as well. Once all the implications are true, this means that all the clauses are true, and thus, the original formula is satisfied.

Of course, the original propositional formula may be always false. In this case, no matter what truth values we plug in, we will always get false. Therefore, once we get the values "true" and "false" from chemical computations, we must check whether they make the formula true. If they do, we return these values; if they do not, we return the message that the original formula was not satisfiable.

A precise description of Matiyasevich's chemical computer: first example. To analyze the behavior of Matiyasevich's chemical computer, they wrote down – and analyzed – the corresponding system of chemical kinetic equations. For simplicity, they assumed that the chemical reactions corresponding to each implication has the exact same intensity.

Before we describe a general formula, let us describe these chemical kinetic equations on the example of the above simple propositional formula. In these equations, we will denote concentrations of each substance v_i by c_i, and concentration of the "opposite" substance $\overline{v}_i$ by c_{-i}.

None of the above five chemical reactions consumes v_1 and two reactions produce v_1: the reactions $v_2 \to v_1$ and $\overline{v}_2 + \overline{v}_3 \to v_1$. According to chemical kinetics, the rate of the first reaction is proportional to c_2 and the rate of the second reaction is proportional to the product $c_{-2} \cdot c_{-3}$. Thus, the differential equation describing the changes in the concentration c_1 of the substance v_1 has the form

$$\dot{c}_1 = c_2 + c_{-2} \cdot c_{-3}.$$

For the substance $\overline{v}_1$, the opposite is true: none of the reactions produces this substance, but we have three reactions that consume it: $\overline{v}_1 \to v_2$, $\overline{v}_1, \overline{v}_2 \to v_3$, and $\overline{v}_1, \overline{v}_3 \to v_2$. The rate of the first reaction is c_{-1}, the rate of the second reaction is $c_{-1} \cdot c_{-2}$, and the rate of the third reaction is $c_{-1} \cdot c_{-3}$. Thus,

$$\dot{c}_{-1} = -c_{-1} - c_{-1} \cdot c_{-2} - c_{-1} \cdot c_{-2}.$$

For the substance v_2, we have one reaction that produces it: the reaction $\overline{v}_1, \overline{v}_3 \to v_2$, and one reaction that consumes it: the reaction $v_2 \to v_1$. Thus, we get

$$\dot{c}_2 = c_{-1} \cdot c_{-3} - c_2.$$

Similarly, we have

$$\dot c_{-2} = c_{-1} - c_{-1}\cdot c_{-2} - c_{-1}\cdot c_{-3};\ \ \dot c_3 = c_{-1}\cdot c_{-2};\ \ \dot c_{-3} = -c_{-1}\cdot c_{-3} - c_{-2}\cdot c_{-3}.$$

From this system of equations, it is easy to see why the chemical reactions will lead to values v_i that satisfy the original formula. Indeed, in these reactions, the substance v_1 is only produced and never consumed, and the substance $\overline{v}_1$ is always consumed and never produced. Thus, after a sufficiently long time, the concentration of the substance v_1 will becomes larger than the concentration of the substance $\overline{v}_1$. According to our interpretation, this means that we will select v_1 to be true.

Similarly, the substance v_3 is only produced, and the substance $\neg v_3$ is only consumed, which means that the substance v_3 will prevail – i.e., that we will select x_3 to be true as well.

We cannot make a similar conclusion about v_2 without performing detailed computations, but we do not actually need to perform these computations: if we select v_1 and v_3 to be true, then, no matter what value we select for v_2, both clauses are satisfied and thus, the original formula is satisfied.

A precise description of Matiyasevich's chemical computer: second example. The conclusion is not always as simple and as straightforward as in the above example. For example, for a formula $(v_1 \vee v_2)\,\&\,(\overline{v}_1 \vee \overline{v}_2)$, by trying all four possible combinations, we can see that it has two possible solutions:

- $v_1 =$"true" and $v_2 =$"false"; and
- $v_1 =$"false" and $v_2 =$"true".

The corresponding equations of chemical kinetics take the form

$$\dot c_1 = c_{-2} - c_1;\ \ \dot c_{-1} = c_2 - c_{-1};\ \ \dot c_2 = c_{-1} - c_2;\ \ \dot c_{-2} = c_2 - c_{-2}.$$

According to our interpretation, what we are really interested in whether $c_1 > c_{-1}$ and whether $c_2 > c_{-2}$. From this viewpoint, it makes sense to consider the differences $\Delta c_1 \stackrel{\text{def}}{=} c_1 - c_{-1}$ and $\Delta c_2 \stackrel{\text{def}}{=} c_2 - c_{-2}$: for each i, we select v_i to be true if $\Delta c_i > 0$ and to be false if $\Delta c_i < 0$.

By subtracting the above expressions for the rate changes of the concentrations c_i and c_{-i}, we can get the expressions for the rate changes of the differences Δc_i:

$$\Delta\dot c_1 = -(c_2 - c_{-2}) - (c_1 - c_{-1});\ \ \Delta\dot c_2 = -(c_1 - c_{-1}) - (c_2 - c_{-2}),$$

i.e.,

$$\Delta\dot c_1 = -\Delta c_1 - \Delta c_2;\ \ \Delta\dot c_2 = -\Delta c_1 - \Delta c_2.$$

By adding these two equations, we conclude that for $\Delta \stackrel{\text{def}}{=} \Delta c_1 + \Delta c_2$, we get $\dot\Delta = -2\Delta$, hence $\Delta(t) = \Delta(0)\cdot\exp(-2t)$. When $t \to \infty$, we get $\Delta(t) \to 0$; thus, for large t, we have $\Delta(t) \approx 0$. By definition of Δ, this means that $\Delta c_1 + \Delta c_2 \approx 0$, i.e., that $\Delta c_2 \approx -\Delta c_1$. Thus:

- If $\Delta c_1 > 0$, i.e., if v_1 is true, then we should have $\Delta c_2 < 0$, i.e., v_2 should be false.
- Vice versa, if $\Delta c_1 < 0$, i.e., if v_1 is false, then we should have $\Delta c_2 > 0$, i.e., v_2 should be true.

So, for this formula, chemical kinetic equations lead to both solutions; which one we get depends on the initial conditions.

A precise description of Matiyasevich's chemical computer: general formula. In general, similar ideas results in the following formula for the rate which the concentration c_a of each literal changes:

$$\dot{c}_a = \sum_{C:a\in C}\left(\prod_{b\in C,b\neq a} c_{\overline{b}}\right) - c_a\cdot \sum_{C:\overline{a}\in C,|C|=3}\left(\sum_{b\in C\,\&\,b\neq\overline{a}} c_{\overline{b}}\right) -$$

$$c_a\cdot \#\{C:\overline{a}\in C\,\&\,|C|=2\}.$$

Here, C goes over all the clauses, $|C|$ is the number of literals in the clause, and $\#S$ is the number of elements in the set S.

Indeed, a substance corresponding to the literal a is produced if a belongs to a clause. Each such clause $a\vee b\vee c$ leads to the chemical reaction $\overline{b}+\overline{c}\to a$ and thus, to the term $c_{\overline{b}}\cdot c_{\overline{c}}$ in the expression for $\dot{a}$.

Similarly, a substance corresponding to the literal a is consumed if the negation $\overline{a}$ belongs to a clause. Each such clause $\overline{a}\vee b\vee c$ leas to the chemical reaction $a+\overline{b}\to c$, and thus, to the term $-c_a\cdot c_{\overline{b}}$ in the expression for $\dot{a}$. If the negation $\overline{a}$ belongs to a clause $\overline{a}\vee b$, then the consuming chemical reaction is $a\to b$, which leads to the term $-c_a$ in the expression for $\dot{c}$.

A simplified version (corresponding to catalysis). In the above system of chemical reactions, each substance is both produced and consumed. To make the analysis of the resulting system of equations simpler, is may be desirable to avoid consumption and consider only production. We can do this if we introduce a new universal substance U and, to each implication $a, b\to c$, assign a modified chemical reaction $U+a+b\to a+b+c$. In this reaction, the input substances a and b are not consumed: in chemical terms, they play a role of *catalysts* that enhance the transformation of the universal substance into the generated substance c.

In principle, in this case, we should also take into account the changes in the concentration of substance U. To maximally simplify the situation, we assume that we have a large (practically unlimited) supply of the substance U, so that the consumption of U during our reactions is negligible in comparison with its original concentration. In this case, we only need to take into account production of each substance, and the resulting differential equations take a simplified form:

$$\dot{c}_a = \sum_{C:a\in C}\left(\prod_{b\in C,b\neq a} c_{\overline{b}}\right).$$

This simplification makes perfect sense from the logical viewpoint:

- In chemical kinetics, a reaction $a+b \to c$ means not only that c is produced but also that a and b are consumed.
- In contrast, in logic, an implication $a, b \to c$ means that if we have some reasons to believe in a and b are true, this increases our belief in c, but it *does not* mean that we somehow decrease our beliefs in a and b.

The above modification of the original system of chemical kinetics equations allows us to avoid this discrepancy.

Simplified equations: example. Let us give an example of such simplified equations. For the above propositional formula $(v_1 \vee \overline{v}_2) \,\&\, (v_1 \vee v_2 \vee v_3)$, the corresponding equations of chemical kinetics take the following simplified form:

$$\dot c_1 = c_2 + c_{-2} \cdot c_{-3};\;\; \dot c_{-1} = 0;\;\; \dot c_2 = c_{-1} \cdot c_{-3};\;\; \dot c_{-2} = c_{-1};\;\; \dot c_3 = c_{-1} \cdot c_{-2};\;\; \dot c_{-3} = 0.$$

Chemical computations implementing Matiyasevich's idea are too slow. Yu. Matiyasevich is a star of the mathematical world, he has a distinction of having solved one of the 23 famous Hilbert's problems – 23 important problems that, at the 1900 World Congress of Mathematics, the 19th century mathematics proposed as a challenge for the next 20th century. Whatever Matiyasevich writes is therefore taken seriously by mathematicians and computer scientists. Immediately, Yuri Gurevich, one of the world leaders in theoretical computer science, engaged his colleagues in the analysis of Matiyasevich's idea: how efficient is it?

Alas, the results of this analysis, published in [1], were not very promising: even for simple propositional formulas, for which simple algorithms produce satisfying propositional values $v_1, \ldots, v_n$, Matiyasevich's system requires exponential time to converge to a correct solution – i.e., to concentrations c_i and c_i for which the vector $v_1, \ldots, v_n$ for which v_i is true if and only if $c_i > c_{-i}$ is indeed satisfying.

Natural idea: let us use high-concentration chemical reactions instead. Since the original chemical reactions are too slow, we need to speed them up. The reaction rate is proportional to the product of the concentrations. Thus, to drastically speed up the reaction, we need to drastically speed up the concentrations c_i and c_{-i}.

The interesting thing is that when the concentrations become very high, the formulas for the rate of chemical reaction change. Indeed, the usual formulas of chemical kinetics are based on the natural idea: that when concentrations are small, then, for the reaction to take place, all the molecules have to physically meet. For example, for a reaction $a + b \to c$ to take place, we need the molecules of a and b to meet.

The total number of molecules of a is proportional to the concentration c_a of the substance a. For each molecule of a the probability of meeting a molecule of b is proportional to the concentration c_b of the molecules b. Thus, the total number of reactions per unit time is proportional to the product $c_a \cdot c_b$ of these concentrations.

In the case of very high concentrations, the molecules are there already, so the reaction always takes place. The rate of this reaction is thus proportional to the total number of pairs (a, b).

- If the concentration c_a of the substance a is higher, then the rate is determined by a concentration c_b of the substance b.

- If the concentration c_b of the substance b is higher, then the rate is determined by a concentration c_a of the substance a.

We can describe both cases by saying that the reaction rate is proportional to the minimum $\min(c_a, c_b)$ of the two concentrations.

This argument may be not absolutely clear when presented on the example of chemical kinetic where we do not have much of an intuition, but it can be made clearer if we use an example of similar predator-prey equations. When the concentrations of rabbits and wolves are small, the rate with which wolves consume rabbits is proportional to the product $c_w \cdot c_r$ of the concentration of wolves c_w and the concentration of rabbits c_r. Indeed, in this case, a wolf has to run around the forest to find his rabbit meal.

On the other hand, if we place all the wolves and all the rabbits together – in a small area where rabbits cannot run and cannot hide – then each wolf will immediately start consuming a rabbit – provided, of course, that there are enough rabbits for all the wolves. So, if the number of rabbits is larger than the number of wolves, the reaction speed will be determined by the number of wolves – hence, by the concentration of wolves c_w: each wolf eats a rabbit. In the opposite situation $c_w > c_r$, when there are more wolves than rabbits, this rate will be proportional to the concentration of rabbits: each rabbit is being eaten by a wolf. In both cases, the reaction rate is proportional to $\min(c_w, c_r)$.

Resulting equations. The main difference between usual chemical kinetics equations and equations corresponding to high concentrations is that we now have minimum instead of the product. Thus, by applying this high-speed high-concentration kinetics to the (simplified) chemical reactions emerging from a propositional formula, we get the following system of differential equations:

$$\dot{c}_a = \sum_{C : a \in C} \left(\min_{b \in C, b \neq a} c_{\overline{b}} \right).$$

For our example of a propositional formula $(v_1 \vee \overline{v}_2) \,\&\, (v_1 \vee v_2 \vee v_3)$, we thus get the following equations:

$$\dot{c}_1 = c_2 + \min(c_{-2}, c_{-3}); \quad \dot{c}_{-1} = 0; \quad \dot{c}_2 = \min(c_{-1}, c_{-3}); \quad \dot{c}_{-2} = c_{-1};$$

$$\dot{c}_3 = \min(c_{-1}, c_{-2}); \quad \dot{c}_{-3} = 0.$$

Discrete-time version of these equations have already been shown to be successful in solving the propositional satisfiability problem. How good is a new system of differential equations? Is it indeed faster than the original one?

The ideal answer to this question would come if we could actually find the substance that have these chemical reactions. Alas, finding such substances is difficult, so we have to restrict ourselves to simulating this system of equations on a computer.

In order to simulate a system of differential equations $\dot{x}_i = f_i(x_1, \ldots, x_n)$, we can use the fact that the derivative $\dot{x}_i$ is defined as a limit of the ratios $\dfrac{x_i(t + \Delta t) - x_i(t)}{\Delta t}$. By definition of the limit, this means that when Δt is small, the ratio is approximately equal to the derivative:

$$\frac{x_i(t + \Delta t) - x_i(t)}{\Delta t} \approx f_i(x_1, \ldots, x_n),$$

hence $x_i(t + \Delta t) = x_i(t) + \Delta t \cdot f_i(x_1(t), \ldots, x_n(t))$.

Thus, if we know the values $x_i(t)$ for some moment of time, we can use this formula to compute the values of all the variables x_i in the next moment of time $t + \Delta t$. Based on the values $x_i(t + \Delta t)$, we compute the values $x_i(t + 2\Delta t)$, etc. If we start at a moment t_0 and we are interested in the values of x_i at the moment t_f, then we need $k = \dfrac{t_f - t_0}{\Delta t}$ iterations of this procedure.

For our system of equations, this means that once we know the values of the concentrations at each moment of time, we can compute the new values of the concentrations as

$$c'_a = c_a + \Delta t \cdot \sum_{C: a \in C} \left(\min_{b \in C, b \neq a} c_{\bar{b}} \right).$$

We repeat this iterative procedure many times, and then select each variable v_i to be true if and only if $c_i > c_{-i}$.

Interestingly, we get the exact same formulas that were proposed by Sergey Maslov in 1980 [12]; see a detailed description and analysis in [10, 11]. In particular, in [10, 11], it was shown that (in contrast to the original Matiyasevich's equations) Maslov's method performs very well on many classes of propositional formulas. For example, for many classes of propositional formulas for which efficient algorithms are known, Maslov's method also comes up with a solution in feasible time.

Thus, we arrive at the following conclusion:

Conclusion. *The use of high-concentration chemical computations is indeed an efficient approach to hard-to-solve problems.*

Historical comment. Maslov's method was originally proposed on a purely heuristic basis, without mentioning chemical computing. The high-concentration interpretation of this method – providing an explanation of why these formulas are used, and a physical justification of why this method should be faster than, e.g., Matiyasevich's approach – is described in [3, 4, 5, 7, 8].

Pragmatic comment. Since Maslov's method was known before, what do we gain by finding out that it coincides with the result of fast chemical computing?

- First, we gain a new justification: Maslov's method is heuristic, and to be able to explain its formulas and explain why they work fast is an advantage.
- We also gain the possibility to naturally modify the original method – e.g., by applying it to the original system of chemical reactions instead of the reactions with a universal substance U – and maybe to find a modification which will work even faster.
- Third, we gain an understanding of how to optimally select a parameter of the Maslov's method: since this interpreted as an integration step of the system of differential equations, we can use known techniques to optimally select this step; see below.
- Fourth, we gain an ability to extend Maslov's technique to problems beyond propositional satisfiability – as long as these problems can be naturally interpreted in terms of chemical processes. For example, in [3, 4, 5, 8], a similar approach was used to find so-called stable models of logic programs. The main difference between a propositional formula and a logic program is that in a formula, an implication $a, b \to c$ automatically leads to $a, \neg c \to \neg b$ – this is why we used three implications and three chemical reactions for each clause; in a logic program, this is not automatically true: rules involving negations have to be explicitly formulated. This difference is easy to describe in chemical computing terms: just add only the rules of the original logic program as chemical reactions (and not the extra rules). As a result, we get an efficient way of computing stable models of logic programs.
- Finally, last but not the least, if we find actual substances that have these chemical reactions, then, by performing these reactions, we can actually solve hard-to-solve problems.

Auxiliary result: how to select the parameter Δt. In our chemical computing model, we start with some concentrations c_i and c_{-i} of the substances corresponding to v_i and $\overline{v}_i$. For these arbitrary concentrations, selecting each propositional variable v_i to be true when $c_i > c_{-i}$ will not, in general, lead to the values that satisfy the original propositional formula F. In the process of chemical reactions, the original inequalities $c_i > c_{-i}$ and $c_j < c_{-j}$ change and eventually, the process (hopefully) *stabilizes* in the sense that the differences $\Delta c_i = c_i - c_{-i}$ no longer change sign. Once the process stabilizes, there is no need to perform further simulations, we can already find the appropriate values of the propositional variables v_i.

Let us denote, by T, the time that it takes for a process to stabilize. In terms of T, if we select a value Δt, then we will need $T/\Delta t$ iteration steps to find the desired solution to the original propositional satisfiability problem. The smaller Δt, the more iterations we need; from this viewpoint, if we want to find the solution faster, we must choose the largest possible value Δt. However, we cannot take Δt too large: otherwise, a linear approximation $x_i(t) + \Delta t \cdot \dot{x}_i$ will not be a good approximation for $x_i(t + \Delta t)$. So, we need to select the largest Δt for which the error of this linear approximation is not too large.

The above linear approximation can be viewed as the sum of the first two terms of the Taylor expansion

$$x_i(t+\Delta t) = x_i(t) + \Delta t \cdot \dot{x}_i(t) + \frac{1}{2} \cdot (\Delta t)^2 \cdot \ddot{x}_i(t) + \ldots$$

The approximation error of the linear approximation is equal to the sum of all the terms that we ignored in this linear approximation, i.e.:

$$x_i(t+\Delta t) - (x_i(t) + \Delta t \cdot \dot{x}_i(t)) = \frac{1}{2} \cdot (\Delta t)^2 \cdot \ddot{x}_i(t) + \ldots$$

In this expansion, each term is (for sufficiently small Δt) much smaller than the next one. Thus, the first (quadratic) term in the right-hand side provides a good approximation for the size of the approximation error. So, to make sure that this approximation error is small, we should require that it does not exceed a certain given portion $\delta > 0$ of the linear approximation, e.g., that

$$\left\| \frac{1}{2} \cdot (\Delta t)^2 \cdot \ddot{x}_i \right\| \le \delta \cdot \|\Delta t \cdot \dot{x}_i\|,$$

where $\|a_i\| = \sqrt{a_1^2 + \ldots + a_N^2}$ denotes the length of a vector $a = (a_1, \ldots, a_N)$. From this inequality, we can find the largest value of Δ for which this inequality is still satisfied, i.e., the largest value of Δt for which linear approximation still works well, as

$$\Delta t = 2 \cdot \delta \cdot \frac{\|\dot{x}_i(t)\|}{\|\ddot{x}_i(t)\|}.$$

In our case, the variables x_i are concentrations c_a corresponding to different literals a. We already have the formula for the first derivatives of these variables:

$$\dot{c}_a = \sum_{C: a \in C} \left(\min_{b \in C, b \neq a} c_{\overline{b}} \right).$$

To find the formula for the second derivatives $\ddot{c}_a$, we need to differentiate the expression for $\dot{c}_a$. A (minor) problem here is that this expression contains minimum. For each t, the minimum of several terms coincides with the smallest of these terms. Thus, the derivative of the minimum is simply equal to the derivative of the smallest term. This leads to the following formula

$$\ddot{c}_a = \sum_{C: a \in C} \dot{c}_{\overline{b}_{C,a}},$$

where $b(C,a)$ denotes the literal for which the value $c_{\overline{b}}$ is the smallest value among all $b \in C$ that are different from a. Once we know the values $b(C,a)$, we can compute the value $\dot{c}_{\overline{b}_{C,a}}$ by using the above general formula for the first derivative $\dot{c}_a$.

As a result, we select the value

$$\Delta t = 2 \cdot \delta \cdot \frac{\|\dot{c}\|}{\|\ddot{c}\|},$$

where

$$\|\dot{c}\| \stackrel{\text{def}}{=} \sqrt{\sum_a (\dot{c}_a)^2} \text{ and } \|\ddot{c}\| \stackrel{\text{def}}{=} \sqrt{\sum_a (\ddot{c}_a)^2}.$$

3 The Resulting Method for Solving Hard Problems Is Related to Numerical Optimization, Neural Computing, Reasoning Under Uncertainty, and Freedom Of Choice

Relation to optimization: why it is important. The fact that Maslov's method turned out to be equivalent to fast chemical computations is nice, but this fact only shows that this method is faster than other *chemistry-motivated* methods of solving the propositional satisfiability problem. Chemical computing has a clear advantage when implemented in vitro – that we drastically parallelize computations. However, if we simply simulate the corresponding chemical reactions on a computer, then there is no convincing reason to restrict ourselves to algorithms that come from such simulations. Instead, we should search for the methods which are the fastest among *all* algorithms, chemistry-motivated or not.

In such a search, we can use the experience of computational mathematics. We cannot *directly* use this experience, because propositional satisfiability – and probably any other NP-complete problem – is *not* something we normally solve in numerical methods. This absence of hard problems from the numerical methods experience makes perfect sense:

- Numerical methods are designed to solve *feasible* problems like optimization (when it is feasible), solving systems large systems of equations or solving a system of ordinary differential equations, problems in which, in principle, an algorithm is known, but because of the large size of the problem, we need to find a faster modification. For these problems, it is possible to find general modifications which allow us to solve the original problems much faster.

- In contrast, for hard-to-solve (NP-complete) problems, there is no general feasible algorithm, solving each of these problems requires creative thinking. Thus, there is no hope (unless P=NP) that we can find a general feasible modification for solving these problems faster.

Since we cannot use a *direct* experience of solving the original propositional satisfiability problem, we must therefore use – *indirectly* – the experience of solving more traditional numerical problems. We have already mentioned that we can use the experience of solving systems of differential equations. Let us now show that we can also use an experience of solving optimization problems.

Relation to optimization: main idea. [7, 16] In the propositional satisfiability problem, we need to find truth values of all the literals a that make the formula $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$ true, i.e., that makes all the clauses $C_1, \ldots, C_m$ true.

In the computer, everything is represented as 0s and 1s. In particular, a truth value is represented as 0 or 1: "true" corresponds to 1, and "false" corresponds to 0. Let us denote, by c_a, the truth value of a literal a. If the literal a is true, then its negation $\overline{a}$ is false, and vice versa; in both cases, we have $c_a + c_{\overline{a}} = 1$.

A clause $a \vee b \vee c$ is true if and only if at least one of the three literals a, b, and c is true. (Similarly, a clause $a \vee b$ is true if at least one of the two literals a and b is true.) To make it easier to compare with the chemical approach, in which each clause leads to equations $\overline{a} + \overline{b} + U \to \overline{b} + \overline{c} + a$ that mostly contain negations, let us reformulate the above condition in terms of negations: a clause $a \vee b \vee c$ is true if and only if at least one of the literals $\overline{a}$, $\overline{b}$, and $\overline{c}$ is false. In terms of truth values $c_{\overline{a}}$, $c_{\overline{b}}$, and $c_{\overline{c}}$ this means that a clause is true if at least one of the non-negative values $c_{\overline{a}}$, $c_{\overline{b}}$, and $c_{\overline{c}}$ is equal to 0. This, in turn, is equivalent to requiring that the minimum $\min\limits_{a \in C} c_{\overline{a}}$ of these values is equal to 0.

In general, we want the (non-negative) expressions $\min\limits_{a \in C} c_{\overline{a}}$ corresponding to all the clauses C to be equal to 0. This is equivalent to requiring that the sum J of all these expressions is equal to 0, where we denoted

$$J \stackrel{\text{def}}{=} \sum_C \left(\min_{a \in C} c_{\overline{a}} \right).$$

It is possible that the original formula does not have any satisfying propositional values $v_1, \ldots, v_n$. In this case, the value J will never become equal to 0. Thus, we can reformulate the original problem as follows: find the values $c_a \in \{0, 1\}$, with $c_a + c_{\overline{a}} = 1$ for all a, for which the expression J attains its minimum. If this minimum is 0, then we get satisfying values v_i. If this minimum is not zero, this means that the original propositional formula cannot be satisfied.

We have reduced the original propositional satisfiability problem to a *discrete* optimization problem, in which the set of possible values of each variable c_a is discrete: it actually consists of two values 0 and 1. This is not exactly what we wanted:

- our goal was to use the experience of numerical methods;
- however, in general, discrete optimization problems are at least as hard as NP-complete problems (see, e.g., [9, 14]); thus, they are not usually solved by numerical methods.

So, to use the desired experience, we must reduce the above *discrete* optimization problem to a *continuous* one. In the above formulation, this can be easily done: just replace each discrete range $c_a \in \{0,1\}$ by a continuous range $c_a \in [0,1]$. Thus, we arrive at the following problem: find the values $c_a \in [0,1]$, with $c_a + c_{\overline{a}} = 1$ for all a, for which the expression J attains its minimum.

Now we can use the experience of numerical optimization. There exist many techniques for minimizing a function $f(x_1,\ldots,x_n)$. Most of these techniques use derivatives of the minimized function f. Among the techniques which only use the first derivatives of f, the fastest is the *gradient descent* method, in which, at each iteration, we replace the original values x_i with new values

$$x_i' = x_i - \lambda \cdot \frac{\partial f}{\partial x_i},$$

for an appropriate value λ.

In principle, we could directly use this formula to the above function J, by computing

$$\frac{\partial J}{\partial c_a} = \lim_{\Delta\to 0} \frac{J(\ldots,c_b,c_a+\Delta,c_c,\ldots) - J(\ldots,c_b,c_a,c_c,\ldots)}{\Delta}.$$

However, this would mean, in general, that we use the values $c_a + \Delta, c_a \in \{0,1\}$ that were artificially added to the original values $c_a \in \{0,1\}$. To make these computations more adequate for the original problem, it may be better to only consider values from the original set $\{0,1\}$, i.e., to use the following discrete approximation $\dfrac{DJ}{Dc_a}$ to the partial derivative $\dfrac{\partial J}{\partial c_a}$:

$$\frac{DJ}{Dc_a} \overset{\text{def}}{=} \frac{J(\ldots,c_b,1,c_c,\ldots) - J(\ldots,c_b,0,c_c,\ldots)}{1-0} =$$

$$J(\ldots,c_b,1,c_c,\ldots) - J(\ldots,c_b,0,c_c,\ldots).$$

Then, we can take $c_a' = c_a - \lambda \cdot \dfrac{DJ}{Dc_a}$.

The minimized function J is the sum of several terms t_C corresponding to different clauses C. One can easily check that the discrete derivative of the sum of several terms is equal to the sum of discrete derivatives of each term. For each term t_C of the type $\min(c_a,c_b,c_c)$, we have $\dfrac{Dt_C}{dc_a} = \min(1,c_b,c_c) - \min(0,c_b,c_c)$. Here, all the values c_b and c_c are in the interval $[0,1]$, thus,

$$\min(1,c_b,c_c) = \min(c_b,c_c), \quad \min(0,c_b,c_c) = 0,$$

and therefore, $\dfrac{Dt_C}{dc_a} = \min(c_b, c_c)$. So, we conclude that

$$c'_{\overline{a}} = c_{\overline{a}} - \lambda \cdot \sum_{C: a \in C} \left(\min_{b \in C, b \neq a} c_{\overline{b}} \right).$$

Since $c_a + c_{\overline{a}} = 1$, any term subtracted from $c_{\overline{a}}$ means that an equal term is added to c_a, so we have:

$$c'_a = c_a + \lambda \cdot \sum_{C: a \in C} \left(\min_{b \in C, b \neq a} c_{\overline{b}} \right).$$

This is exactly the chemical kinetics formulas, with λ instead of Δt.

Relation to numerical optimization: conclusion. We can conclude that *the chemistry-motivated formulas for solving hard-to-solve problems can also be justified by the experience of numerical optimization.*

Relation to numerical optimization: what do we gain from it? We can ask the same pragmatic question that we asked before: what did we gain by this optimization justification? Well, first, we gained a new justification, but – similar to the previous section – there are more pragmatic gains as well:

- First, we now use the experience of numerical optimization to come up with a new method for selecting $\Delta t = \lambda$ (and for checking whether the selected parameter Δt is adequate). Namely, we can estimate the quality of each iteration c_a if we normalize the corresponding values to the condition $c_a + c_{\overline{a}} = 1$, by taking $\widetilde{c}_a = \dfrac{c_a}{c_a + c_{\overline{a}}}$ and computing the value $J(\{\widetilde{c}_a\})$ of the minimized function. If the value of J on the next iteration is larger than the value on the previous iteration, this means that we moved too fast, and we should decrease the value λ; numerical optimization techniques recommend halving λ. Vice versa, if the value J on the next iteration is smaller, this means that maybe we can move faster, so we can try doubling λ and seeing what happens.

- Another idea is that instead of gradient methods that only use the first derivatives, we can use faster second-order methods that use second derivatives as well; see, e.g., [2, 16, 17].

Relation to neural computing. Neural computing is a way to perform computations by simulating how such computations are performed in the human brain. In the human brain, the state of each neuron is usually well represented by a real number – the frequency with which this neuron generates pulses. When the neuron is active, it generates a lot of pulses; when the neuron is inactive, it generates only a few pulses. Neurons send these pulses to other neurons, and the received signal changes the state of receiving neurons.

In the first approximation, we can say that a neuron can be in two states: active and inactive. For example, a neuron receiving optical signals from the eye is active if there is light coming to the corresponding portion of the eye, and inactive if there is no light coming to this portion. Similarly, when we think about an object or a person, certain neurons are activated. Thus, it makes sense to assume that when we think about how to solve a propositional satisfiability problem with propositional variables $v_1, \ldots, v_n$, it makes sense to assume that to each variables, there is an appropriate neuron that becomes active when we have reasons to believe that this variable is true, and inactive if there are no such reasons.

In the brain, frequently, two different neurons (or groups of neurons) correspond to each property. For example, when we are asleep, neurons that are normally very active are de-activated, but, on the other hand, other neurons – who are normally not active at all – become very active. So, it makes sense to assign neurons also to negations $\overline{v}_i$: such neurons becomes active when $\overline{v}_i$ is true (i.e., when v_i is false). Thus, we assign a neuron to each literal a.

Each rule $\overline{b}, \overline{c} \rightarrow a$ means that if we have reasons to believe in $\overline{b}$ and in $\overline{c}$, then this gives us extra reasons to believe in a. In neural terms, this means that if the neurons $\overline{b}$ and $\overline{c}$ are active, then the neuron a also becomes more activated, i.e., we add a term to the original activation level c_a. This term is added only when both neurons are activated, i.e., when $c_{\overline{b}} > 0$ and $c_{\overline{c}} > 0$. Similarly to our analysis of the optimization relation, we can show that this combined condition is equivalent to $\min(c_{\overline{b}}, c_{\overline{c}}) > 0$. Thus, it makes sense to add to the original activation level c_a, for each implication of the type $\overline{b}, \overline{c} \rightarrow a$, a term proportional to this minimum.

As a result, when we take into account all the implications corresponding to all the clauses, we get the same Maslov's formula as in the case of chemical computing. Thus, this formula can also be interpreted in neural terms.

Comment. Minimum $\min(a, b)$ is, of course, not the only function with the property that it is positive only if both a and b are positive; we can therefore try other such functions as well. In particular, Maslov himself proposed to use functions $f_r(a, b) = (a^r + b^r)^{-1/r}$ for $r > 0$. When $r \rightarrow \infty$, these functions tends to $\min(a, b)$. When using these functions instead of minimum in the iterative method of solving the propositional satisfiability problems, he also got very good results; the justification of using this family of functions is given in [7].

Historical comment. S. Maslov himself presented this heuristical neural derivation of his iterative method in numerous talks, but he never published it. The details of Maslov's derivation were first published in [15].

Relation to reasoning under uncertainty. In the traditional mathematical reasoning, each statement is either tree of false. In reasoning under uncertainty – e.g., in reasoning about expert knowledge – it is important to take into account that we may have different degree of confidence in different statements. For example, a medical expert can say that a bleeding large-size irregularly shaped skin tumor is probably cancerous, but this expert understands well that this statement is sometimes false.

A natural idea is to represent this degree of certainty by a number from the interval $[0, 1]$, so that:

- a complete certainty – meaning that the statement is true – corresponds to 1,
- the absence of any argument in favor of this statement (meaning probably that this statement is false) corresponds to 0, and
- intermediate degrees of certainty are represented by numbers from the interval $(0, 1)$.

We may have different arguments in favor of a statement a and in favor of its negation $\overline{a}$; in this case, when we need to make a definite decision:

- we select a if our confidence c_a in the statement a is larger than our confidence $c_{\overline{a}}$ in its negation $\overline{a}$, i.e., if $c_a > c_{\overline{a}}$;
- we select $\overline{a}$ if our confidence in the negation $\overline{a}$ is larger, i.e., if $c_{\overline{a}} > c_a$.

In these terms, an implication $a, b \to c$ means that if we believe in both a and b, then we have additional reason to believe in c, i.e., that our degree of certainty in c increases. Arguments similar to the neural case show that it is reasonable to add, to the degree of certainty of a, a term proportional to $\min(c_b, c_c)$ (or to $f(c_b, c_c)$ for some other combination function). Thus, we also arrive at Maslov's iterative formulas.

Relation to freedom of choice. Freedom of choice was the original motivation of Maslov's iterative method – it is explicitly mentioned in the title of his first paper [12] describing this method; see [6, 11] for details.

This idea is easy to explain: Initially, we have a large search space, whose size grows exponentially with the length of the input. For example, for propositional satisfiability with n Boolean variables $v_1, \ldots, v_n$, this search space includes 2^n possible combinations of "true" and "false" values. Because of the huge size of this space, we cannot test all its elements. Instead, we must test only a few "most possible" candidates for a solution. For example, for propositional formulas, we can cut the size of the search space in half if we fix a value of one of the Boolean variables v_i to a certain value ε_i ("true" or "false").

Since we are not testing all the elements of the search space, we may miss a solution. So, we must select a subclass with the smallest "probability" of losing a solution. In particular, for propositional satisfiability, we must select a variable v_i and a value ε_i for which the probability of losing the solution is the smallest possible. After each choice (v_i, ε_i), there may be several solutions.

If we knew exactly the number of solutions $N(v_i, \varepsilon_i)$ left after each choice, then we could simply take a solution for which $N(v_i, \varepsilon_i) > 0$. In reality, however, we do not know these values $N(v_i, \varepsilon_i)$. At best, we know the *estimates* $\widetilde{N}(v_i, \varepsilon_i)$ for these numbers.

Usually, we have no information about the errors $\widetilde{N}(v_i, \varepsilon_i) - N(v_i, \varepsilon_i)$ of these estimates. Therefore, it is natural to assume that larger values of error are less probable than smaller ones. Hence, the larger the estimate $\widetilde{N}(v_i, \varepsilon_i)$, the larger the probability that for this choice (v_i, ε_i), the actual number of solutions will be positive, and therefore, that we will not miss a solution.

As a result, a reasonable method is to look for a choice (v_i, ε_i) after which the estimated number of solutions $\widetilde{N}(v_i, \varepsilon_i)$ is the largest possible. In other words, we must make a choice after which *the remaining freedom of choice is the largest possible.* Maslov called this idea "the strategy of increasing the freedom of choice".

Let us denote the estimate $\widetilde{N}(v_i, \varepsilon_i)$ by c_i if $\varepsilon_i =$"true" and by c_{-i} when $\varepsilon_i =$"false.

Each clause $a \vee b \vee c$ can be reformulated in the form $\neg a \& \neg b \to c$. From the viewpoint of the *freedom of choice* strategy, this means that if, according to our estimate, there are many solutions for which $\neg a$ and $\neg b$ are true, then the estimate for the number of solutions for which c is true must also increase. By arguing like in the neural case, we conclude that for the corresponding estimates c_a, we get exactly Maslov's iterations.

Thus, Maslov's iterative formulas can be justified based on freedom of choice as well.

Comment. While Maslov's method prompted by this freedom of choice principle is new, the principle itself have been formulated, in various forms, by different researchers. For example, David Marr, a well-known researcher in computer vision, described a similar principle as the *Principle of Least Commitment.*

Acknowledgments

This work was supported in part by the National Science Foundation grants HRD-0734825 and DUE-0926721, by Grant 1 T36 GM078000-01 from the National Institutes of Health. The authors are thankful to Evgeny Katz for his encouragement.

References

[1] A. Blass and Yu. Gurevich, "On Matiyasevich's nontraditional approach to search problems", *Information Processing Letters*, 1989, Vol. 32, pp. 41–45.

[2] R. I. Freidzon, M. I. Zakharevich, E. Ya. Dantsin, and V. Kreinovich, "Hard problems: formalizing creative intelligent activity (new directions)", *Proceedings of the Conference on Semiotic aspects of Formalizing Intelligent Activity, Borzhomi, Republic of Georgia, 1988*, Moscow, 1988, pp. 407–408 (in Russian).

[3] L. O. Fuentes, *Applying Uncertainty Formalisms to Well-Defined Problems*, Master's thesis, Department of Computer Science, University of Texas at El Paso, 1991.

[4] L. O. Fuentes and V. Ya. Kreinovich, "Simulation of Chemical Kinetics as a Promising Approach to Expert Systems", *Abstracts of the Southwestern Conference on Theoretical Chemistry*, El Paso, Texas, November 1990.

[5] L. O. Fuentes and V. Kreinovich, *A touch of Mexican soul makes computers smarter*, University of Texas at El Paso, Department of Computer Science, Technical Report UTEP-CS-91-6, 1991.

[6] V. Kreinovich, "S. Maslov's Iterative Method: 15 Years Later (Freedom of Choice, Neural Networks, Numerical Optimization, Uncertainty Reasoning, and Chemical Computing)", In: V. Kreinovich and G. Mints (eds.), *Problems of Reducing the Exhaustive Search*, American Mathematical Society, Providence, Rhode Island, 1996, pp. 175–189.

[7] V. Kreinovich, "Semantics of S. Yu. Maslov's iterative method", In: *Problems of Cybernetics*, Moscow, 1987, Vol. 131, pp. 30–62; English translation in: V. Kreinovich and G. Mints (eds.), *Problems of Reducing the Exhaustive Search*, American Mathematical Society, Providence, Rhode Island, 1996, pp. 23–51.

[8] V. Kreinovich and L. O. Fuentes, "Simulation of chemical kinetics – a promising approach to inference engines, In: J. Liebowitz (ed.), *Proceedings of the World Congress on Expert Systems, Orlando, Florida, 1991*, Pergamon Press, New York, 1991, Vol. 3, pp. 1510–1517.

[9] V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl, *Computational complexity and feasibility of data processing and interval computations*, Kluwer, Dordrecht, 1997.

[10] V. Kreinovich and G. Mints (eds.), *Problems of Reducing the Exhaustive Search*, American Mathematical Society, Providence, Rhode Island, 1996.

[11] S. Yu. Maslov, *Theory of Deductive Systems and its Applications*, MIT Press, Cambridge, Massachesutts, 1987.

[12] S. Yu. Maslov and Yu. N. Kurierov, "Strategy of increasing the freedom of choice when recognizing propositional satisfiability", *Abstracts of the All-Union Conference "Methods of mathematical logic in artificial intelligence and system programming"*, Vilnius, Lithuania, 1980, Part 1, pp. 130–131 (in Russian).

[13] Yu. Matiyasevich, "Possible nontraditional methods of establishing unsatisfiability of propositional formulas", In: *Problems of Cybernetics*, Moscow, 1987, Vol. 131; English translation in: V. Kreinovich and G. Mints (eds.), *Problems of Reducing the Exhaustive Search*, American Mathematical Society, Providence, RI, 1996, pp. 75–77.

[14] C. H. Papadimitriou, *Computational Complexity*, Addison Wesley, Boston, Massachusetts, 1994.

[15] O. Sirisaengtaksin, L. O. Fuentes, and V. Kreinovich, "Non-traditional neural networks that solve one more intractable problem: propositional satisfiability", In: *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, Atlanta, Georgia, May 28–31, 1995, Vol. 1, pp. 427–430.

[16] M. I. Zakharevich, In: *Problems of Cybernetics*, Moscow, 1987, Vol. 131; English translation in: V. Kreinovich and G. Mints (eds.), *Problems of Reducing the Exhaustive Search*, American Mathematical Society, Providence, Rhode Island, 1996.

[17] M. I. Zakharevich, In: *Proceedings of the Conference on Semiotic aspects of Formalizing Intelligent Activity, Borzhomi, Republic of Georgia, 1988*, Moscow, 1988, pp. 141–145 (in Russian).