WHY STEM?

Kosheleva, Olga, PhD, Associate Professor
Kreinovich, Vladik, PhD, Professor
University of Texas at El Paso
olgak@utep.edu, vladik@utep.edu

Abstract: Is the idea of combining science, technology, engineering, and mathematics into a single STEM complex a fashionable tendency, as some educator think – or is there a deep reason behind this combination? In this paper, we show that the latest developments in Theory of Computation make such a union necessary and desirable.

Keywords: STEM, theory of computation, feasible algorithms, NP-complete problems

Why STEM? In the past, teaching math and teaching science were different topics, somewhat related, but largely independent. Nowadays, however, it is fashionable to consider teaching all STEM disciplines – science, technology, engineering, and math – as a whole.

But why? Is there a real reason why, e.g., science and math should be studied together – or it is simply fashion, and in a few years, a new trend will replace this one?

In this paper, we show that there is actually a good reason to combine different disciplines, and this reason comes from the latest developments in theory of computation.

What is theory of computation: a brief reminder. To explain the relation, let us briefly describe what theory of computation is about; see, e.g., [1, 2].

Computations research is, by definition, an applied discipline. The ultimate goal of designing new computers and new algorithms is to solve real-life problems.

Whenever a researcher comes up with a new algorithm, he or she tries to make it as general as possible, so that this algorithm can be used in other (similar) situations as well. This tendency can be traced in all aspects of human activity:

- People did invent a car just to go from a specific place A to a specific place B: the purpose of the car is to bring us from any point to any other point – as long as there is a road.
- Similarly, Edison did not just invent a lamp to light his own desk – his lamps could be used everywhere.

This desire for generality is reasonable but, if we generalize too much, we end up with a problem which may be impossible to solve at a current technological level. A car that travels along any road is possible to make, but if we want it also to connect points across the ocean (or even get to any place on the Moon), that would doom the car-design project.

Because of this danger, it is very important to find out which problems can be solved and which cannot be solved – so that we do not doom our design project from the very beginning by making it impossibly complex. In terms of algorithms, this means that we need to find our when problems can be algorithmically solved and when they cannot. This is one of the main tasks of theory of computation.

This is a difficult task, and it is made even more difficult if we take into account that the mere existence of an algorithm is not enough: if this algorithm requires several billions years to run on a reasonable-size input, this algorithm is of no practical use. What we need is not just algorithms, we need algorithms which are *feasible.*

What is a feasible algorithm: discussion. This natural and seemingly simple question does not have a good answer yet. What *is* clear is that for a given algorithm, whether this algorithm is feasible or not depends on its *worst-case complexity* t(n), i.e., on the largest number of its computational steps for all inputs of size n. If this complexity increases too fast when the size n increases, the algorithm is not feasible; otherwise, it is feasible.

If the time t(n) grows exponentially, e.g., as 2^n, then even for reasonable size inputs n = 300, the number of computational steps becomes so huge that even if we assume that each computational step takes the smallest physically possible time – e.g., the time during which the world's fastest possible process (light) goes through the smallest possible object (elementary particle) – we will still take much more time than the lifetime of the Universe. Clearly, such algorithms are not feasible.

On the other hand, for most known feasible algorithms, the time t(n) is bounded by some polynomial of n: e.g., by a linear function Cn, or by a quadratic function Cn^2, or by a cubic function Cn^3.

What is a feasible algorithm: resulting definition. These example led to the current formal definition of feasibility: *an algorithm is called feasible if there exists a polynomial* P(n) *for which* t(n) < P(n) *for all* n.

This definition is not perfect. The above definition sounds very reasonable, so why did we say that there is no good answer yet? Because a deeper analysis shows that this definition is not perfect.

- Indeed, e.g., $t(n) = 10^{100}n$ is clearly a polynomial, but clearly not feasible.
- On the other hand, the function $t(n) = \exp(10^{-100}n)$ grows faster than a polynomial, but it is clearly feasible for all realistic sizes n.

The current formal definition of feasibility is not perfect, but it is the best we have – so we will use it in this text.

What is a "problem"? As we have mentioned earlier, the main purpose of an algorithm is to solve problems. So, the next natural question is: what is a problem? To answer this question, let us describe problems from different disciplines.

A professional *mathematician* proves theorems. To a mathematician, a problem is:

- given a statement x,
- find a proof of either the original hypothesis y, or of its negation ~y.

Once a proof is presented, it is feasible to check it – of course, this requires it to be a full proof, not a sketch filled with statements "it is not that difficult to prove that". Once we have a full proof, all we need to do is check each step. Humans can do it, even computers can do it – already in the 1960s, when computers were much slower, they were successfully used to check the proofs. In other words, we have a feasible checking algorithm C(x, y), that:

- given x and y,
- checks whether y is indeed a proof – i.e., whether y is indeed a solution to the original problem.

Another requirement is that for a proof to be meaningful, it cannot be too long. This was the problem with the first computer-generated proofs: they were so long that no one could check them. What does ``not too long'' mean? Same as what feasible computations mean – that the length len(y) of y cannot exceed some polynomial of the length of x: len(y) < L(len(x)), for some polynomial L(n).

What do *physicists* do? In a nutshell:

- they are given the data x, and
- they want to find a law y that explains all this data.

This is what Newton did when we came up with a formula that explains all gravity effects, this is what Ohm did when he came up with his famous Ohm's law. Once the law is found, it is relatively easy to check that all the given data fit this law – it may be cumbersome but it is definitely doable, and computers can sure do it. So, in this case, we also have a feasible algorithm C(x, y) for checking whether y is a solution. We also have a limitation of the length of y: namely, we must have len(y) < len(x), because otherwise, we can simply list all the data x and call it a law.

What do *engineers* do?

- They are given specifications x, e.g., specifications for a bridge, and

- they need to find a design y that satisfies all these specifications.

Nowadays, checking that all specifications are satisfied – e.g., that the bridge will not collapse under a hurricane-strength winds – is feasible: just run a simulation program. This does not mean that engineer's work is easy:

- while checking a design is relatively easy,
- coming up with the design is often very difficult – especially taking into account that there are usually severe constraints on the cost.

Here also, there is a limitation on the length of y, this time caused by the fact that this design has to be practically implemented.

These and other examples lead to the following formal definition of a problem.

A formal definition of a (general) problem. *A general problem can be defined as a pair* (C,L*), where* C *is a feasible algorithm, and* L *is a polynomial.*

An instance of this problem is:

- *Given a string* x
- *Find a string* y *for which* C(x, y) *is true and* len(y) < L(len(x)).

Terminological comment. In theory of computation, general problems have a special (somewhat weird) name, motivated by the fact that in this theory, in addition to normal (deterministic) computational devices like finite automata, it is often useful to consider fictitious ``non-deterministic'' devices, in which as part of a computation, we are allowed to make guesses.

In these terms, solving a general problem is a good example of non-deterministic polynomial-time computations:

- once we have guessed a solution y,
- we can check, in polynomial (= feasible) time that our guess is indeed a solution.

As a result, the class of all general problems is denoted NP, short for Non-deterministic Polynomial.

Additional comment. Are there problems outside the class NP? Definitely.

One example is *optimization* problems, when we are not just looking for *a* design that satisfies the given specification, but we are instead looking for *the cheapest* of such designs. There is often no easy way to check that the presented design y is indeed the cheapest. The only practical way is to try all possible designs – but there are usually astronomically many of them.

Another case is *games* like chess. What we want in such games is not a simple design, but a strategy that will enable us to win no matter what moves the opponent makes. There is no easy way to check that, short of trying all possible opponent's strategies – and there are unrealistically many of them.

So yes, there are real-life problems outside the class NP. However, most real-life practical problems are in NP.

Which problems are tractable and which are not? Some problems from the class NP can be solved by a feasible (i.e., polynomial-time) algorithm. For example, if we have a list of numbers, we can sort it in increasing order by using a simple polynomial-time algorithm. The class of all such *tractable* problems is usually denoted by P, short for Polynomial-time.

Interestingly, no one was able to prove that there exist problems which are not in the class NP, i.e., that the class NP is indeed different from its tractable subclass P. This is the famous "is P = NP" question.

Most computer scientists, however, believe that these two classes are different, so we will follow this belief in this paper. While we *do not know* whether any of the problems from the class NP are more difficult than P, we *do know* that some problems from the class NP are the most difficult in this class – namely, that every problem from the class NP can be reduced to this particular problem. Such problems are known as *NP-complete*, or *intractable.*

General problems of mathematics, physics, engineering are all provably intractable. Many practice-related general problems have been proven to be NP-complete (intractable). For example,

- while there exist feasible algorithms for solving systems of linear equations,
- solving systems of quadratic equations is already known to be intractable.

Similarly, general problems of:

- mathematics (proving theorems),
- physics (finding a law that matches all the data),
- engineering (finding a design that satisfies all the specifications) –

all these general problems have been proven to be NP-complete.

What is the practical consequence of this NP-completeness? OK, we have a nice and somewhat gloomy theoretical result. Can be conclude anything from this result other than that life is tough?

Actually, we can. Remember that NP-completeness of a problem means that every other problem from the class NP can be reduced to this problem. So, if we have an efficient algorithm for solving one NP-complete problem, then, by reducing all other problems to this one, we can get a feasible algorithm for solving all other problems.

For example, if we are good in solving math problems, we can thus solve problems from physics and from engineering. This is not as counter-intuitive as it may seem: this is what scientists have been doing for centuries: using math to solve physical problems.

Vice versa, if we are good in solving physics problems, we can thus solve problems from mathematics. This may sound even less intuitive, but again, engineers have been doing this for decades: to solve a mathematical problem, we can emulate it by real physical processes. Such analog computers have indeed been very successful.

So we need STEM! All this bring us back to the original question: do we need STEM combination? Is it a natural combination? In pragmatic terms, does it help us solve problems?

As we can see from the above test, our answer is enthusiastically *Yes!*

- Once we have mastered solving mathematical problems, due to NP-completeness, we this gain a new way to solve problems in physics and engineering.
- Once we have mastered solving physics problems, due to NP-completeness, we this gain a new way to solve problems in mathematics and engineering.
- Once we have mastered solving engineering problems, due to NP-completeness, we this gain a new way to solve problems in physics and mathematics.

It thus makes sense to study all these subjects in coordination, emphasizing the relation between them, emphasizing how they help each other.

On second thought, did we say anything new? Such inter-twining of physics, mathematics, and engineering is already well known and well used in teaching: for example,

- some calculus problems are easier to solve by symbolic manipulations, while
- other problems become easier to solve if we realize that these seemingly abstract mathematical problems can be reformulated in intuitively understandable physical terms.

Similarly:

- some physical problems are easier to solve by considering the corresponding physics, while
- for other problems, it is easy to write down and solve the corresponding equations.

So, did we say anything new? Or did we use complex notions from theory of computation to confirm that two plus two is four?

Yes, we did. At first glance, it may sound as if theory of computation just provides a fancy and unnecessarily complex way to saying what we knew from the very beginning: that, e.g., math and physics can help each other.

But a deeper analysis of NP-completeness shows that its consequences go way beyond this simple conclusion:

- Yes, we know that *sometimes*, reformulating a math problem in physics terms (or vice versa) helps. In teaching terms, this means that sometimes, it is nice to cross-reference the corresponding classes.
- On the other hand, NP-completeness results means that once we have an efficient algorithm for solving problems from one class, we automatically get a new algorithm for solving problems from other class as well. In other words, by using a slight exaggeration, we can say that reformulating a math problem in physics terms (or vice versa) *always* (and not just sometimes) helps – at least potentially. In teaching terms, this means that it makes sense to consider teaching mathematics, physics, and engineering together, as a single entity.

What needs to be done. We have shown:

- that the STEM combination is based on a natural idea,
- that this combination is potentially very helpful to all the related disciplines, and
- that there are indeed examples of such help.

What we need to do is to explore this potential even more, to provide new examples showing the students how their progress in one of these disciplines help them solve problems in all other related areas.

Acknowledgments. This work was supported in part by the US National Science Foundation grant HRD-1242122 (Cyber-ShARE Center).

The authors are thankful to Dr. Mourat Tchoshanov for his encouragement and valuable discussions.

References

1. Kreinovich, V., Lakeyev, A., Rohn, J. and Kahl, P. Computational Complexity and Feasibility of Data Processing and Interval Computations, Kluwer, Dordrecht, 1998.

2. Papadimitriou, C. H., Computational Complexity, Pearson, Boston, Massachusetts, 1993.