From an implementation detail to a new understanding of the intertwining between theory and real-life computing

– breaking a secret mass spell –

Daniel Porumbel, CEDRIC-CNAM

June 23, 2022

I have, undisturbed thereby, pursued the train of my thoughts for more than thirty years, and precisely only because I had to and could not do otherwise, from an instinctive drive that was yet supported by the conviction that what truth an individual has thought and what obscurity he has illuminated will yet at some time also be grasped by another thinking spirit, will speak to it, give it pleasure, and console it [note: even if at a very late moment]; such a one we are addressing, just as those similar to us have addressed us and thereby been our consolation in this living wasteland.

Arthur Schopenhauer

This work aims at shedding light on the difficulties that arise on the contact point between theory and real-life computing. It is often on this contact point that a good part of the intellectual energy is wasted when implementing a mathematical algorithm. The translation of a higher-level programming language into machine code is less critical, because it can be quite efficiently automatized by modern compilers. The most difficult is to cross the border between the realm of theory and the realm of programming. The life of a daily cross-border commuter who does that every single day can be (much) harder than many may think. Why that? Because living in these two realms require quite different intellectual skills and sometimes almost contrary scientific callings; moreover, an adaptation effort is needed each time one crosses the border, to make the brain switch from one working mode to another.

Furthermore, many people very attached to theory have a very schematic understanding of real computing and do not want to hear too much about it. This is particularly true if they see programming as one of those little things – a necessary evil – that have to be done in order to make algorithms work. As long as this view remains in power, we are under the grip of a kind of systemic spell that fractures the community in two:

(a) On the one hand, many theory expert authorities *who have rarely seen running algorithms* may lack the insights needed to have a complete understanding of practical computing. For example, their discourse often focuses on the "big ideas"", without realizing that the dynamics and efficiency of an algorithm can easily change if one modifies a "small" parameter or even if one introduces a redundant constraint in the model. I'm afraid that they are often happy if they understand practical computing in very broad lines. And it is impossible to get to the root of such a question if you study it from the heights of a (simplifying) theory. Let us not deceive ourselves: whoever has never struggled with a computer to control the dynamics of an algorithm will never know the secrets of computing beyond certain simplifying generalities and axioms. And experience has taught me an another law of human nature: the more sterile a theory expert is, the more chances he has to be too arrogant to ever change his ideas.

(b) On the other hand, real-life programmers *who see running algorithms almost every day* do not always have – especially the very young ones – the most appropriate cognitive tools, analysis patterns and thinking models to detect and avoid certain traps and misleading values of the overall system. This prevents them from developing a truly clear and complete vision of the place and value of their work in a more general (scientific) context.

These non-technical ideals project in thousands ways on the technical understanding of practical and theoretical computing. On has to choose either (a) or (b) more often than it may seem.

How many students find themselves disarmed and unprepared as to the best way to code the algorithms of the PhD adviser when they start their PhD? Let's consider also the master courses usually offered in Operations Research (OR) or Computer Science (CS). It is usually not difficult to find many courses on programming techniques, compilation and software engineering. There are also many courses on different theories in various fields of mathematics, OR or CS. But, in my opinion, there are not enough courses that try to "marry" theory and real computing. I do not simply think of computer classes where students simply implement algorithms learned in a theory course; that is very common in practice. I think about courses with a title showing an explicit interest in the contact points between theory and practice.

Let's look over slide below, taken from a talk at ISMP 2018 (by A.V. Goldberg):

```
What students learn?
* Theory of Algorithms
    - Ignore constant factors for machine independence
    - Analysis mostly worst-case
* Programming Languages
    - Aim for compiler to take care of all low-level details
    - High-level specification
* Software Engineering
    - Correctness
    - Development Speed
    - Portability, maintainability and testability
RED: Tempting to ignore constant factors and low-level details
```

This list of courses can be extended to cover endless concepts that need to be studied; for instance, in OR, we can study valid inequalities, decompositions, facets-inducing cuts, (meta-)heuristics, etc. My feeling is that (too) many professors prefer to focus their teaching on the "big ideas", disconnecting them too much from the programming layer (that some forgot almost completely). But can one take the best algorithmic decisions in the everyday life by only taking into account the "big ideas"? For my part, I really do not think so. In any case, there are few courses that try to link the "Theory of algorithms" and "Programming languages" aspects from the above slide. We thus forget to teach the students the art of efficiently coding a algorithm already designed in theory. Once we have proved all the valid inequalities and we have decomposed the model, what's the next step to make it work? Can the results obtained in the programming step be fed back to the theory, like in a system with a feedback loop? The difficulty of designing a course on such aspects confirms that a lot of intellectual energy is lost in translating theory into practice.

Research-wise, I have already met a researcher in theoretical computer science who prides himself of producing only algorithms designed to be never implemented (even if published in top conferences like FOCS/SODA/STOC). He was a real artist, because he did not want to know if his work would ever serve a purpose (visible from the outside world). How different is the spirit of an engineering inventor! For him, theory alone means nothing by itself: it is like love without deeds. Thus, his aim is to give life to theory by setting it into motion, so as to make a computer do something – at worst only crazy things but still something.

Many think that programming should remain a secondary matter simply because theory naturally represents the most noble "fundamental research". This is partly true and partly false; it mainly applies to the best 1% theoretical researchers out there: I accept these people should indeed not be bothered with implementation questions. But the story is not so wonderful or glorious for the other 99% of researchers. They often avoid implementation issues for more mundane reasons: you have to take instances, run endless numerical tests, dive into the code again and again at each revision of the paper; in short, it requires painstaking efforts. As a result, many people say to themselves something along the lines: "I'll look for a little nice theory. And if I can get along well with the few dozen researchers working on it, it will be all great; I'll be so happy with that!."

This may also give one a hard-to-challenge right to call ones work "fundamental theoretical computer science". It is quite complex to counter – or more precisely to unmask – certain people who overstate the importance of their theory field and claim to produce "fundamental theoretical research". It is true that the notion of fundamental research, which may exist legitimately, does not always have a direct and immediate impact on practical aspects. But many have wrongly inverted this argument (in their heads) to draw a fallacious conclusion: everything that is far from any down-to-earth reality and has no practical impact must be "fundamental research". And there is a second artful strategy that is employed more often than one might think.[1] They choose some rather narrower field (a niche area) where they retreat as if in a shelter behind a meticulously-woven web of high-sounding words made to dazzle the reader. It may contain: many overly intricate expressions, new and unheard-of keywords designed to produce increasingly audacious high-sounding effects, excessively long formulae as often as possible, all described in a dialectically exaggerated manner, using an overly-technical jargon. This can dazzle and mislead those who think that anything hard to decode or understand must contain only very powerful and very highly-intellectual ideas. It is easy to be dazzled this way, knowing that is is very difficult for an outsider to see very clearly through such a dense web.

I was very close to falling into (the trap of) such a misleading track. You may notice in my publication list a paper on an algorithm with no software implementation, in the theory of submodular function minimization [1]. At the time, I thought: "The theory of these functions seems cool; it even attracted the attention of the Fulkerson prize jury who awarded the prize (in 2003) for contributions in this area. Everything seems very learned: it is a nice small world sheltered from outsiders". I later learned that the 2021 Abel Prize was awarded (to László Lovász), in part, for work in submodular functions. However, this track might have been a wrong one for me. One day I realized that I was following it only to seek (unconsciously) the shelter offered by working in a niche area inside one of the countless theories humankind has produced. Since I am not one of the 1% top researchers on the theoretical aspects, I realized that my purely theoretical work will never get anywhere close to the Fulkerson prize. And I thus also understood that a more implacable law was likely to govern my life: those who choose such a shelter end up pushed aside, forever forgotten.

I thus learned to avoid purely theoretical CS work to explore an area that touches more people: the intertwining between theory and real-life computing. It should not be understood that I want to run away from theory at all costs. I have written 100 pages on the theory of Semipositive Definite

[1]This strategy is an old one, because it has been used for centuries beyond the world of the exact sciences. Here is how Schopenhauer described it in his field, philosophy: "Now, to conceal a want of real ideas, many make for themselves an imposing apparatus of long compound words, intricate flourishes and phrases, immense periods, new and unheard-of expressions, all of which together furnish an extremely difficult jargon that sounds very learned. [...] I refer to the artful trick of writing abstrusely, that is to say, unintelligibly; here the real subtlety is so to arrange the gibberish that the reader must think he is in the wrong if he does not understand it, whereas the writer knows perfectly well that it is he who is at fault, since he simply has nothing to communicate that is really intelligible, that is to say, has been clearly thought out* Without this device Fichte and Schelling could not have established their pseudo-fame. "

Optimization or SDP [2], a theory more complex than that of linear (integer) programming. But I only think to find a way to connect it to real-life computing one day. I am not trying to use it, for example, to design theoretical approximation algorithms (e.g., Goemans-Williamson); that is out of question for me.

An important and very subtle difficulty is to learn how to combine theory and programming, without moving too far away from either of them. And the best way to keep them together is to make them leave inside a single brain. This task becomes more and more difficult with age. The older you get, the more reluctantly you do the programming work. But I think it is important to overcome this obstacle. *A work produced by a composite brain, where one person develops the theory and another person does the coding* can only have a secondary importance in my heart.

I described on purpose this practice as if it were a question of private sensibilities, to be clear that I will express a purely personal opinion here. I think that this practice, which cuts the links between theory and programming, leads to a *fractured and emotionally-hardened community* that evolves like a child in an orphanage: even the most brilliant technical service cannot overcompensate the demoralization that results from the lack of real interest in programming. But this artificial obstacle can be disregarded: by its ability to set real computers in motion, *the notion of uniting theory and programming inside a single brain will one day find all its meaning*, leading it to a dignity superior to that of many theoretical niche areas.

Why be so sure? Because theory and programming are so intertwined and interrelated that their harmony can only be achieved if we manage to make them form a solid couple that lives inside a single brain, without ever separating. It is not enough to meet once every 10-15 days to tell each other what you have done; this breaks the harmony of the couple. I came to this conclusion from experience, not from reading a book. The best theoretical decisions I have ever made have always been the result of direct knowledge of the (subtle) algorithmic phenomena at work. And to gain full insight into the finest algorithmic phenomena, a theoretical understanding may not be enough; we rather need *a theoretical understanding illuminated and set into motion by the countless findings offered by practice and programming.* If you think this is obvious, that everyone knows that the theory-programming couple must live in one brain, show me thousands of people who live this credo. On the contrary, most publications are the result of a composite brain: some people take the role of theoretical guides and they outsource the coding tasks to others.

The root of this problem is related to the above mentionned fractured state of the CS community. It's easy to find very strong theoretical scientists. It's not too hard to find very good software developers either. And many could be very strong on both tasks. But too many *choose* not to try such a double endeavour – often because they failed to see the interest in it.

I think the best way to master the essence of an algorithm with any practical calling is to implement it yourself. A typical example is Local Search (LS). It looks easy to read and understand. But if you try to implement it only using some broad theoretical knowledge acquired very rapidly, you will get very poor results on any competitive problem. LS may seem simple, but it takes months and months of both practice and theory to turn it into a competitive solution method for a highly studied problem. If you do not do this, the potential of a composite brain that ignores the links between theory and programming may not remain competitive in the long run. The phenomenon can go beyond hard sciences. I admit that it may depend a lot on the context, but there are many domains in which the value of a composite brain may be more limited than you noticed. Literature is a good example. Take your favorite book. It was written by one person, with rare exceptions.

A further obstacle is that it is rather hard to sell the notion of a theory-programming couple living hapily in a single brain. In such a fight, you may be both victorious and defeated before you even start. I explain the paradox. Such struggle is lost in advance because our system will never finance or honour this kind of idea which has (apparently) no grandeur and nothing bombastic. The narrative in its favour may even give the impression of returning to very simple and basic elements, to a kind of past that many have left "behind". It's quite hard to describe it in very modern colors, or even to slip in a buzzword or a bombastic phrase.

But each time you decide to engage in a line of work that is not honoured by the system, you will always find a little something that can be gained before you even start. The level of greatness or tininess is not essential. In such a struggle, one always finds the satisfaction of taking part in a fight of a free man against the man absorbed by the system, against the man that simply adheres to some mass standards. The free man is at least sure not to grow up forever like "another brick in the wall". More generally, such struggle also comes with a satisfaction that we often forget too easily: it is the scarcity that gives the product its value, *i.e.,* only what is rare can be valuable. The men absorbed by the system are so numerous that they can become invisible in the mass; if they can hardly stand out from the crowd, they will easily be forgotten in the last instance. This will be even more true if one day our (scientific) system begins to fall under the weight of an increasingly large mass of people who care about the rewards of adhering to the system infinitely more than about truth seeking. If this ever reaches a too large extent, who will still take them all (lemmings) seriously?

These ideas are not meant to suggest advice to everyone without discrimination; I am only giving personal and general opinions. And I repeat that most of what I said does not apply to the top 1% theoretical researchers who have a real calling for their theory and should not be disturbed with practical implementation problems. But I hope that my ideas will one day be useful to other people, to all those who feel today perhaps as lost as I was 10 years ago. I first speak, a bit in the spirit of the motto of this paper, to all freelancers out there who are looking for a new relationship with theory.

I have already discussed related ideas with various colleagues at Roadef and elsewhere. Many of them often agreed on different points. But I felt that the essence of my ideas evaded them. It was towards the completion time of my work that I had the pleasure of finding an ally in the past history of the university of Oxford, in the person of Christopher Strachey (1916-1975). Actually, I'm not sure I can call him my ally. But I felt that he took the right words out of my mouth. The things he wrote 50 years ago gave me great confidence and consolation; I hope this can be repeated many times in the future – in the spirit of the motto of this paper. Here is the full text still displayed on-line at `www.cs.ox.ac.uk/activities/concurrency/courses/stracheys.html`:

> The strongly held common philosophy of the Department (previously Oxford University Computing Laboratory) is admirably expressed in the following quotation from Christopher Strachey, the founder of the Programming Research Group which now forms part of Computing Science at the Department:
>
>> It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen.

Bibliography

[1] Daniel Porumbel, Prize-Collecting Set Multi-Covering With Submodular Pricing, *International Transactions in Operational Research*, 25 (4):1221-1239, 2018
[2] Daniel Porumbel, Demystifying the characterizations of SDP matrices in mathematical programming, report technique CNAM, `http://cedric.cnam.fr/~porumbed/papers/sdp.pdf`
[3] Daniel Porumbel, D'un « petit détail » d'implémentation vers un manifeste sans limite (sur l'éloignement entre la programmation et la théorie), `cedric.cnam.fr/~porumbed/soft/`