A. Verschoren
University of Antwerp
Department of Mathematics and Computer Science
Antwerp, Belgium
alain.verschoren@ua.ac.be

Towards a computer science program: adventures of a mathematician in the world of informatics

1. Introduction

When I received the invitation to participate to the workshop *"High standards in research and teaching in computing engineering and related subjects – some reflections"* organized by José Luis Freire in A Coruña, I was flattered but also somewhat surprised. Of course, I have been knowing José Luis for many years and even if I have been acquainted with certain subjects within computer science, I still remain a mathematician, even a rather pure mathematician. From this point of view, I might definitely not be the best placed person to answer fundamental questions about good practices within the field of computing engineering. It was only later, when glancing through the list of participants and speakers, that I realized that the aim of the meeting was actually to bring together people with different background and experience, to interact on the subject and thus to attain extra insight and ideas based on this confrontation.

To say the least, even the subject itself of the meeting was somewhat controversial. It has been clear for decades that the relationship between computer science and software engineering is somewhat unclear, even at the level that it remains undecided which topics software engineering and computer science are (supposed to be) dealing with. Specialists claim that computer science should be studying the properties of computation in general, whereas software engineering should con-

centrate on the design of specific computations to achieve practical goals. A definition which is far from perfect, but which has at least the merit that it separates the subject(s) into two separate and complementary disciplines.

One should also not forget that computer science, in its broadest sense, has at least two academic roots, one situated within mathematics and with strong connections to logic and computation theory, the other within electrical engineering with somewhat more emphasis on hardware aspects. These ambivalent roots do not make things much simpler, although it remains clear that computer science departments usually find their origin in engineering-based departments, whereas computational science departments remain much more related to mathematics departments. Moreover, both types of department invest serious efforts in trying to align their education and research.

2. First steps

How does one develop a computer science curriculum? This might appear to be a theoretical, academic[1] question, but at our university, it was a very concrete one, back in the eighties.

Maybe some background should be included to understand the situation. The University of Antwerp used to be composed of three independent sub-universities. First there was RUCA (the "Rijksuniversitair Centrum Antwerpen"), a state university, which had its origin in an economic highschool which was started in 1852. It offered undergraduate courses in sciences and medecine, and had a complete faculty of applied economics. On the other hand, there was UFSIA (the "Universitaire Faculteiten Sint Ignatius Antwerpen"), a jesuit university, also having its origin in an economic highschool, started at about the same period. It offered undergraduate courses in humanities, and had a complete faculty of applied economics, in direct competition with its RUCA analogue. Finally, there was UIA (the "Universitaire Instelling Antwerpen"), a 'pluralistic' university, which only started in 1971 and

1 Academic: theoretical or hypothetical; not practical, realistic, or directly useful: *an academic question; an academic discussion of a matter already decided*.

was aimed at graduates and PhD's, both in sciences and humanities. Education at UIA was very innovative at the time, based on a trimester system, a credit system, a limited number of contact courses but a definite emphasis on active education and self-study, a system that is nowadays widely adapted in Flanders.

These three universities were fused into a single one (the "UA", the University of Antwerp) in 2003, after years of dialogue – a non-trivial task in view of the competitition between both Faculties of Applied Economics and the different 'color' of RUCA and UFSIA (state versus private-catholic).

We had a department of mathematics at UIA in those days, the good old days, where at the graduate level several options were possible: pure mathematics, applied mathematics, mathematical physics and computer science.

Of course, even then, all undergraduate students in mathematics had to follow some courses in computer science, but this essentially amounted to elementary programming, some basic ideas about computability or Turing machines and the use of computing techniques within applied mathematics. And that was it, except when you chose to continue at the graduate level, still within mathematics, in the direction of proper computer science.

At the undergraduate level, we did not have genuine computer science at that time, but we offered degrees in dentistry, that is, until a fire destroyed two of the main labs at RUCA. This forced us to make some choices. One possibility was to rebuild the labs and reinvest in dentistry, but dentistry was not really a very successful direction at our university – how many dentists does a country need? Another option was to stop offering courses in dentistry and exchange this for something else, and this was exactly what we did. You have to know that our university was by Belgian law an incomplete university - we were not allowed to offer courses in civil engineering or psychology, for example - and, by the same law, we could only start a new direction in exchange for another one.

So RUCA decided on an undergraduate degree in computer science. We started it within the mathematics department, before making it an independent degree,

mainly to be compatible with what happened at UIA, where computer science remained an option within mathematics.

3. A program?

A program had to be developed. Within a department of mathematics and, ergo, by mathematicians and a minority of already present computer scientists. But even these were mainly dealing with rather theoretical topics: they worked on topics like theoretical databases, formal grammars or logic. We of course hired extra staff, but highly in function of the program we were developing. A nice program, but a complete fiasco: the percentage of students that failed the first year was large. No, it was huge! The reason was probably a lack of efficiency and disinformation: most students had a pc at home, they were accustomed to use Microsoft products, and sometimes they also had some elementary ideas about Basic or Pascal. And that was about it. They had a complete misconception about what computer science was about ("Computer science is no more about computers than astronomy is about telescopes") or: they just wanted to write programs (but had no idea about the algorithms behind it). Moreover, at that time (and even nowadays!) computer science was poorly taught at secondary schools, classes limiting themselves to teaching students how to use specific programs.

But they, our future students, thought that they were already well prepared – and we did not really contradict them. On the other hand, stemming from a mathematics department, we thought that we should include a lot of math in the first and second year programs. Of course: all of us, while thinking about the "ideal" computer science program had taken a look at what we thought was some kind of computer science bible: Knuth's marvelous book [1], a book, or a series of books, dealing with the fabulous applications of mathematics in computer science. A book full of number theory, Fibonacci sequences, formal power series and so on, definitely fundamental mathematical tools which appeared to be essential if one wanted to write his or her first computer program!

All this seemed marvelous at the time: we offered students computer science from within a department of mathematics, and we could continue teaching what we taught before - algebra, analysis, operator theory and so on - but now with a defi-

nite insight: even pure mathematics has become widely applicable with the rise of of computer science. What we did not realize at the time was that our students and future students maybe did not exactly know what computer science was about, but that there was one thing which they definitely did not want, at least not too much, and that was: mathematics!

Of course, this was not the only reason our start was a complete fiasco. Not being "genuine" computer scientists, it was very hard for us to judge the exact time that a student would need to spend on the concrete projects we suggested to them. As told before, even at that time, we tried to use innovative educational techniques, including projects and group work. We included this point of view in the computer science curriculum, and tried to complement our rather formal, mathematical approach with lots of individual practical activities, but not being experts, it was really quite impossible for us to judge the time that would effectively be spent on these activities by the students. It should not come as a surprise that the projects were much more time consuming than we expected.

To put it briefly: after our first year's experiment our students learned something (a lot even), but so did we:

4. Lesson 1

- Explain clearly what your topic is about before students start; be honest! Do not try to attract as many students as possible: emphasize that the subject might well be difficult, their previous preparation insufficient and that they will have to work hard!
- Use mathematics as a tool, but do not exaggerate: students interested in computer science are interested primarily in computer science, and not in mathematics, even if mathematics is a marvelous tool (and a nice subject, by the way – as a mathematician, I should know!).
- Try for a good mix between theory and practice, convince the 'hard-liners' that there is nothing wrong with applying their nice theoretical results, convince the 'applied' people that there is nothing wrong in looking for theoretical foundations and tools.

- Do not include too much mathematics, and if you do, explain why (there are lots of connections with logic - refer to Turing machines, Gödel - but also algebra provides for applications).
- Include lots of projects in your program, let them work and learn what it really is all about, but be sure that you know exactly how much time should be spent on these projects.

5. Something about the author

I already mentioned above, in the introduction, that I am a mathematician, even a pure mathematician (sorry!) - I am essentially working within the fields of ring theory and algebraic geometry. The reason I was invited to the meeting probably has something to do with the fact that I have also been dealing with computer science the last two decades.

This was not really 'on purpose': at some moment some colleagues of mine asked me whether I would be interested in contributing to a joint project on "Machine Learning". I said 'yes'. For the wrong reason: I have been interested in new teaching techniques for many years and I thought that the project would deal with computer assisted teaching. Quite on the contrary, it appeared the project was devoted to developing artificial intelligence based techniques to make the computers more 'intelligent', not the user. I only realized this when it was too late, i.e., when I sat face to face with an impressive Sparc Station, one Sun's top nodge machines at the time (this was part of the project), an excellent workstation running unix, and vi as an editor. For the younger readrs: I am speaking about times when X-terminals and graphical interfaces on unix machines were just on the brink of developing.

After the initial shock I indeed learned unix and C and C++ (and vi!) and started to like the field. I read some stuff on AI, quickly grew into evolution based optimization techniques, and found myself after some time within the field of genetic algorithms. Originally I concentrated on theoretical aspects of this subject, but it quickly appeared that the techniques we developed ('we', that is the people with whom I work in our ISLab (Intelligent Systems Lab)) were widely applicable within economy, at the level of datamining, and proteomics. There is still a strong collaboration on the mathematical, theoretical aspects of genetic algorithms with

some colleagues (in particular in La Coruña), but most output is nowadays quite applied and really situated within biochemistry and work on large data-sets.

6. Something about genetic algorithms

The idea behind genetic algorithms is quite straightforward. Indeed, some functions are just too difficult to optimize by ordinary, classical algorithms and tools, for example due to exponential behavior in their growth. Moreover, sometimes we do not really want to find (or: cannot find) 'the' maximum, but just need a reasonable approximation. In situations like this, a solution may be found in the use of evolutionary algorithms, like genetic algorithms, or techniques like simulated annealing, which realize exactly what we need: they do not aim at finding the exact maximum of a function but rather find in a polynomial time reasonable approximations of a maximal solution or even indicate its 'structure'.

Most of these algorithms are based on a competition/cooperation model between possible solutions of the optimization problem. Genetic algorithms work with populations of binary strings representing possible solutions to the underlying optimization problem and let these populations evolve through competition between these strings, based on ideas essentially stemming from Darwinism, including 'surviving of the fittest' and genetic modification like cross-over or mutations.

The (surprising!) fact is that these techniques work, at least if we let the algorithm run 'long enough' and if we include sufficient knowledge. Of course, there remain some fundamental questions when applying a genetic algorithm, like: "How long does it take before we find a reasonable approximation of the solution we are seeking?", i.e., "When do we stop the algorithm?". This question is difficult, let us admit it, and its study involves statistical methods, Markov chains, Brownian motion, differential equations and catastrophe theory, for example.

Another basic question is: "How can we see whether a function is difficult or easy to optimize?", a question which is, of course, tightly connected to the previous one. Here we need to introduce some invariants which help to classify the functions to be optimized into classes with predictable difficulty ('GA-hardness', as it is sometimes called). These invariants include notions like order, deception and epistasis.

The last invariant, epistasis, associates to every function a real number between 0 and 1, which for large classes of functions is an indicator of its level of difficulty, the easy (linear!) functions having epistasis 0 and the more difficult functions having epistasis closer to 1.

To introduce and study epistasis, one needs a large amount of linear algebra, convex analysis and differential geometry. To give an example, from linear algebra, one needs serious input dealing with:

- matrices;
- diagonalization, eigenvalues and spectra;
- generalized (Moore-Penrose) inverses;
- abstract vector spaces (in particular basis manipulation);
- tensor algebra;
- Walsh-Hadamard transforms (this is some kind of discrete Fourier transform used in image compression, for example).

It appeared that, although the applications of these genetic algorithms are highly applicable in concrete situations, one needs a serious mathematical background, but also, within and without mathematics, a wide diversity of complementary techniques.

When I started working on these topics, it was unclear which part of my math background would come into the picture. I was unable to guess or predict where the (possible) applications should or could be situated. Even more: to be able to apply some of these techniques outside of their theoretical realm, one needs some interdisciplinary background or better, at least someone who is able to translate problems from biology, chemistry or economics into our language, be it mathematics or computer science.

From this I learned a new lesson:

7. Lesson 2

- Take into account that your curriculum is not made for eternity: the quality of your program will not be evaluated by scholars but society and its demands. On the other hand: do not forget that a university is supposed to provide answers to questions which have not been raised yet - show some initiative.
- Stimulate students to think nonlinearly, in a creative way. Stress that there is something like "serendipity" and that they should never believe that there is a straightforward way of going from 'A' to be 'B'. Do not trust mathematicians who claim otherwise!
- Stimulate "freedom of research": include free space in your program to allow students to pick up topics from other disciplines if they want, and even stimulate them. You never know when this comes in handy! Do not force them, however: they wanted to study computer science in the first place.
- Computer science should be treated from an interdisciplinary point of view. This does not mean that all these interdisciplinary aspects have to be included in the curriculum: seek advice from colleagues working in other disciplines when needed.
- But beware:
  - biologists (economists, physicists, chemists, sociologists, ...) sometimes (usually) do not understand what mathematics or computer science is about or what these disciplines can do for them;
  - computer scientists (and mathematicians) sometimes (usually) do not know what biologists are talking about;
  - so: one needs translators, people acquainted with both topics and once the translation is made (this may take some time!), each group can work in his own surroundings;
  - concentrate on team building: it is preferential having a group of experts working together than concentrating on everything but not knowing a lot on anything.
- Do include mathematics (but not too much and maybe not for everybody)!

8. Towards a genuine program

So, having experimented and learned a lot from facts and fallacies, we started thinking and rethinking our program. It had to include and emphasize, but not too much, interdisciplinarity, at least at the master's level. Pedagogically we decided to make our curriculum project-oriented and problem-based: do not teach them before they ask, let them not be confronted with answers before the questions arise naturally and give them the opportunity to make mistakes to learn from. Let us not exclude involvement from industry, even if our program is pending somewhat more on the theoretical side: most of our students will be hired by industry, anyway. And finally, let us include as much flexibility as possible in our program: provide for a quick start of *the* or *a* new curriculum, which should be open to easy modifications; make the program central, not the individual courses (and ego's).

They sometimes say in Flanders: "We moeten het warm water niet opnieuw uitvinden"[2], so we tried not to start from scratch this time but we looked at ideas elsewhere.

In particular, we took into account the *Joint IEEE/ACM Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*, which define the core knowledge areas of computer engineering as consisting of:

- Algorithms
- Computer architecture and organization
- Circuits and signals
- Database systems
- Digital logic
- Digital signal processing
- Electronics
- Embedded systems
- Human-computer interaction
- Operating systems

2 "Let us not re-invent hot water!"

- Programming fundamentals
- Social and Professional issues
- Software engineering
- VLSI design and fabrication
- Computer Networking
- Distributed Systems

Of course, we had to make a choice: although the program should include both theoretical and practical aspects, it is impossible to work these out in full depth. We decided upon a more theoretical approach and tried to combine mathematical foundations, computation and data structures with concrete applications. We tried to include a reasonable amount of optional courses. Students were to be guided by their teachers in their choice of topics. The legal system in Belgium does not permit us, however, to organize genuine "on-demand-teaching": the curricula are supposed to have a reasonable degree of uniformity and students are supposed to have a comparable background when they finish their studies. Of course, there is much more freedom at the graduate than at the undergraduate level.

The courses about computation theory and data structures were rather easy to decide upon and would include topics like, for example, automata, languages and grammars, computability and complexity on the side of computation theory and analysis of data structures or algorithms on the side of data structures. Filling in the mathematics part was surprisingly more complicated. We started with

- logic
- graph theory
- data theory
- category theory
- numerical analysis
- computational geometry
- discrete mathematics

But what about topics like cryptography, numerical algorithms and computer algebra? From my own experience, I recalled that one might also need linear algebra, group theory, number theory, statistics, calculus and even representation theory,

system theory, differential equations, operational research and fuzzy logic. Should all of this be included? Well, even with our math roots, we decided against it. Our point of view was: let us offer the basics and if a student needs a particular topic at some point of his or her career, it is only then that the necessary background should be picked up. On the other hand, we wanted to leave sufficient free space in our program to give our students the opportunity to included topics from other curricula, even outside of the proper faculty of sciences or our own university. In this way, they would be able to follow their own interests and 'invest' in the future.

During these discussions, we learned:

9. Lesson 3

- Make some choices. Theoretical and applied computer science are hard to combine. In particular, theoretical computer science is, in principle, more suitable to be organized within a faculty of sciences, applied computer science within a faculty of engineering.
- Specialize, even within choice/options. There is no harm in making the contents of your masters, even in the Bologna context, depend on the presence of local specialists and local research. Do not try to be the best at everything - you just cannot!
- Evolve. A program is not made for eternity.
- Work together with other universities. Attracting lots of students only realizes part of your funding: more than 50 % of your income comes from research activities. Combine competition and cooperation.

10. A program!

The bachelor part is essentially a scientifically funded introduction to several subtopics within the field of computer science, complemented by necessary ingredients from other fields like mathematics and physics. In includes the possibility to acquire programming skills.

Starting from the first year, there is a strong emphasis on concrete projects and team-work. In the second and the third year, there is already ample opportunity to

put personal accents in the program: more than one quarter of the program - the so-called "profiling space" - can be filled in with optional courses. These may be directed towards a specialization within computer science, as a preparation to a further specialization at the master level, or towards a broadening of the student's scope by including topics not properly connected to computer science, like economics or statistics.

The total amount of "contact hours" is about 600 hours per year. Exercises and applications are dealt with in small groups of students.

Bachelor 1	
Introduction to programming	6
Discrete mathematics	9
Computer systems	6
Languages and automata	6
Data structures	6
Introduction to software engineering	6
Calculus	9
Computer networks	6
Computer graphics	6

Bachelor 2	
Advanced programming	6
Machines and computability	6
Operating systems	6
Databases	6
Numerical linear algebra	9
Algorithms and complexity	6
Project: distributed computing	9
+ 12 to choose from:	
Network applications	3
Physics	6
Programmation paradigms	6
Elementary statistics	3
General economy	3

At the master level, we offer 4 specializations, each of these totaling 120 credits (roughly 600 hours per year):

- Computational computer science
- Computer networks and distributed systems
- Data-bases
- Software engineering

In each of these specializations, 30 credits are compulsory in the first year, 12 may be chosen freely from a list of optional topics. Moreover, the students have to choose between the options *Entrepreneurship,*

Bachelor 3	
Thesis	9
Databases (XML + web technology)	6
Scientific programming	6
Software engineering	6
Computer architecture	6
Telecommunication systems	6
Philosophy and Ethics	3
+ 18 to choose from Bachelor 2 or:	
Introduction to distributed systems	3
Data mining	3
Applied logic	3
Artificial intelligence	3
Advanced programming techniques	3
Formal techniques in software engineering	6
Network protocols	3
Image processing	3
Economy and entrepreneurship	3
Data structures and algorithms	3

Education or *R & D'* (18 credits). The option *Entrepreneurship* is mainly aimed at students who choose for a job position in the business world. *Education* is meant for students who want a teaching position and *R & D* for students who aim at a research career within the university or industry.

The program for the specialization "Computational computer science" looks as follows:

Computational computer science: Master 1	
Compilers	6
Distributed systems	6
Modeling and simulating	6
Optimisation techniques	6
Computational finance	6
+ 12 to choose from (or other discipline)	
Mathematical methods in image processing	3
Computational geometry	3
Multilevel and multi-grid methods	3
Computer linguistics	6
Cluster computing	6
Digital signal and image processing	6
Capita selecta in computational sciences	3
Computational neuroscience: machine learning	3
Wavelets	6
Optimalisation	6
Technical-scientific software	6
Deterministic and stochastic integration techniques	4

Option Entrepreneurship	
Communication	6
Managing an organization	6
Financial management and legal aspects	6
Option Teaching	
Introduction to education	3
Didactics of computer science	6
Practica/students/...	9
Option R&D	
Scientific English	3
Communication	6
Science philosophy	3
Research project I	6

Computational computer science: Master 2	
Thesis	30
+ 12 to choose from (or other discipline)	
Mathematical methods in image processing	3
Computational geometry	3
Multilevel and multi-grid methods	3
Computer linguistics	6
Cluster computing	6
Digital signal and image processing	6
Capita selecta of computational sciences	3
Computational neuroscience: machine learning	3
Wavelets	6
Optimization	6
Technical-scientific software	6
Deterministic and stochastic integration techniques	4

Option Entrepreneurship	
Innovation and entrepreneurship	6
Process management	6
Option Teaching	
Educational management	3
Practica/students/...	3
+ 6 to choose from:	
Class management	3
Teaching 'special' groups	3
Stimulating thought process	3
Language and learning	3
Mathematical didactics	3
Option R&D	
Research project II	6
+ 6 'personalized' extra.	6

Distributed systems: Master 1	
Compilers	6
Distributed systems	6
Mobile and wireless networks	6
Prestation analysis and telecom systems	6
Distributed computing paradigms	6
+ 12 to choose from (or other discipline)	
Advanced performance modeling	6
Lab mobile and wireless networks	3
Grid computing	6
Cluster computing	6
Seminar computer networks	3
Software for real-time and embedded systems	4
Capita selecta in distributed systems	5
Internet infrastructure	5
Multi-agent systems	4
Secure network and computer infrastructure	4

Data-bases: Master 1	
Compilers	6
Distributed systems	6
Advanced data-base systems	6
Recent trends in data-bases	6
Data-mining	6
+ 12 to choose from (or other discipline)	
Mathematical methods in image processing	3
Data-base security	3
Advanced AI techniques	6
Project data-bases	6
Statistical methods and data-analysis	4
Text-based informational retrieval	4
Machine-learning and inductive inference	4
User interfaces	4
Web-information systems	4
Foundations of data-bases	4
Bio-informatics	6

Software engineering: Master 1	
Compilers	6
Distributed systems	6
Modeling and transformation	6
Formal specifications	6
Software re-engineering	6
+ 12 to choose from (or other discipline)	
Capita selecta of software engineering	6
Data-base security	3
Software testing	6
Data-mining	6
Distributed software architectures	6
Requiremants analysis	6
Aspect oriented software development	6
Programming language engineering seminar	6
Software architecture	6

The master thesis corresponds to 9 credits. There are different types: it could be research oriented, it may reflect the result of a theoretical or practical project or build upon results previously published in the literature, for example. The student is usually guided by a team and he or she has to "defend" the thesis at the end of the second (last) year. This defense is usually consisting of a short presentation to a jury, which might include people from outside, e.g., working in industry, in the presence of fellow students. After the formal presentation, some questions may be asked and there will be some interaction about the contents of the thesis.

Acknowledgement

The I wish to thank the organizers for the excellent organization of this workshop. Maybe I should also stress the fact that this note only reflects personal ideas and that the department of mathematics and computer science at the University of Antwerp should not be held responsible for its contents.

References

1. Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.