Software Engineering Without a Clue: the perils and pitfalls of programming graphs you don't really understand.

James Neil,
School of Computer Science, University of Birmingham.
j.r.neil@cs.bham.ac.uk

Introduction

What is *non-classical computation*? What is a non-classical computer? How can we program them? There are lots of examples of non-classical computers and computation, involving processes as diverse as *BZ-chemical reactions* [1], *liquid crystal* [2], and *Genetic Regulatory Networks* (GRNs) [3]. Much research occurs in simulation, but simulated equations and networks lack the speed of a true, physical implementation: hence the interest in novel computational media. The fundamental problem however is 'programming' non-classical computers. Many biological systems are examples of non-classical computation, and are perhaps the only natural computers of any complexity. We recognise, in some sense, that they program themselves; what is fair to say, however, is that we don't really understand how they do this, how we can emulate it, and how we can use it for our own purposes. We have only begun to explore natural computation and biology from a computational standpoint. We must ask ourselves, at the outset, what is it that we should be looking for in the natural world?

The Classical/Non-classical Trade off.

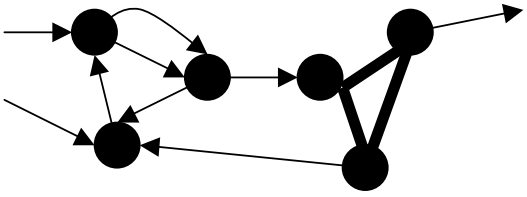
We are all familiar with the *Von Neumann (VN) machine*, the archetype of *classical computation*. It is the physical implementation of the single tape *Turing Machine*, where a Universal CPU executes commands and accesses memory from a *finite* shared sequential storage block. Classical computation is the more general set of computations executed on real or virtual machines that are 'in the spirit' of the VN machine. Compared to non-classical computation, classical computation is easy for us to understand; indeed, the single tape Turing machine was conceived as a device that could do anything that a mathematician could do, given enough time and paper. The closeness of VN model to the process of rigorous human thought and calculation has enabled us to write new intuitive programming languages; affording huge advances in our ability to build and program such devices.

But the cost of this intuition has been parallelism. Non-classical computation represents the class of computations that are not in the spirit of VN machines: they are not sequential, classical, and in general, human understandable. The category is broad and ill defined, comprising in part of what we consider connectionist, natural and/or physical computation. These computations can be parallel, noisy, asynchronous and physical.

For the most part, and probably the whole part, these computations are a subset of the computations we are used to. Assuming, with good reason, that nature cannot compute beyond the limits defined by the classical theory of computation, it is clear that any non VN computation can be simulated, to an arbitrary precision, by a Von Neumann machine. The difference then, must be that non VN computations are naturally implemented on parallel, naturally inspired *non VN machines*.

The Underlying structure of Non-classical Computation

Much of what we consider non-classical, non VN, computation can be easily understood as graphical computation. In this kind of computation information is altered, and then passed from one computational unit to another. These computational units could be as simple as transistors, or as complex as biological neurons, and they operate in parallel, and preferably asynchronously. The computational units, together with their communication mechanisms, form a graphical structure, and the properties of this structure, together with the properties of the computational units, define the computation.



A Computational Graph with one hyper-edge and one pseudo-edge

We must note that the graphical structure need not simply be a traditional graph: it could be a pseudo-graph, with more that one edge possible between vertices, or it could be a hyper-graph, where there are hyper-edges that connect three or more computational units together. Whatever the nature of the graph, it is a clear intuition that, even ignoring the properties of the computational units, the graphical properties of the graph are fundamental to the nature of the computation.

It could be argued that many real world computations cannot be thought of in a connectionist graphical way: though this could be true, it is far from evident, and so should be contested. The graphical structure underlying non-classical computations such as *Neural Networks* and GRNs is clear, and much inter-cellular communication could be modelled using hyper-graphs; it is not clear, however, what is occurring in non-linear materials such as Harding and Miller's Evolvable liquid crystal system [2]. This question of what is happening in such systems is essentially unanswerable until we have a deeper understanding of both non-classical graphically based computation and of non-linear materials. It seems likely that materials shown to have interesting computational properties will be shown to move information to spatially distributed area of computation. The presence of meaningful graphical structure in many non-classical computations is, however, obvious.

If consideration of the graphical structure of non-classical computation is of fundamental importance, then we must address the issue of how we can discover the properties of good *computational graphs*: the set of graphs with structural properties advantageous to computation. However, before we consider this further, let us consider the analogous case for classical, von Neumann computation. We will do this by asking: "How could be program a Von Neumann machine if we didn't have a clue?"

The Art of Classically Bad Computer Programming

Let's consider the behaviour of a truly awful computer programmer: one who doesn't understand even basic syntax. Their programming methodology is simple: type randomly at the keyboard, attempt compilation, and then alter the program; repeating until the code compiles. Obtaining compiling code is the first, and the least, of their problems: the code must now be altered until it performs to specification, using only a hill-climbing strategy, and, most likely, largely uninformative feedback about the quality of their program. All things considered, our coder had better be a fast typist!

Such a coder seems doomed to unemployment; however, with knowledge of programming syntax, they may yet be saved. By having enough knowledge to always write programs that compile, the first step – obtaining a program that complies – can be avoided, and the second step – obtaining a program that fulfils the specification – can be sped up, as any alteration to a program results in a program that will compile.

The prospects for this coder, although better, are still abysmal. They literally do not know what they are doing; they are babbling, and their colourless green for-loops may well sleep furiously. It is clear, in the world of traditional Von Neumann computation and computer programming, you need to have some idea of what you are doing if you are to have any success at all. Anyone using this second coding strategy would quickly fail his or her degree program, which would not be good for their career prospects.

We might suggest to our poor coder that he or she look at some existing computer programs, so that they might gain some idea of the form and structure of computer programs. They might notice that there are lots of loops, but that they are not often deeply nested. They might notice that ratio of if statements to while loops, or the average number and type of variables in each procedure. There are a great many statistics that could be compiled to help guide the 'search' that the coder employs to write their code. With enough information, and a superhuman ability to type, this coder may well have some success.

Non-classical coding: typing really fast at a non-classical keyboard – with a bit of a clue.

We recognise that this is not how we write programs, but then we have a clue – about some things. When we write programs we consciously understand the problem, or at least break it down into problems that we do understand. This is not something that we can do with most non-classical computations. We understand some problems and specifications well enough to hand code operating systems, word processors and flight simulators. We resort to other methods, like Neural Networks, when we want to solve machine-learning problems.

The problem is that we have largely been relying on a strategy similar to the second strategy: for example in graph based GP representations, such as Cartesian Genetic Programming [4], we let fitness selection alone drive the graph's structure. Biological GRNs have a complex of chemistry of gene regulation, which has a large role to play in the structure of the implicit network. This has been largely ignored in computational GRNs, although some work in this area has been done by Bentley [3].

What we lack is a knowledge of graphical structure in non-classical computations. We cannot resort to hand coding these systems, so we must resort to the third coding strategy. To do this we must first understand what the properties of computational graphs are, and how we can tractably capture these properties in algorithms that essentially search for programs by altering the properties of the graph and of the computational units. It is thankful that computers are, effectively, very fast typists.

What should we be looking for?

If we had a large number of computation graphs that we knew to be decent solutions to a number of problems, we could analyse them, just like computer programs, for statistical information. We may be interested in the number of directed loops, their size and their relationships to each other with respect to the smallest chain of computational units needed to reach one from the other. We may be interest in the number of computational on average in a hyper-edge, and the complexity of the relationship between them as a function of the number of computational units. We may have some initial intuitions to guide us, but ultimately, we must statistically analyse collections of graphs for structures that we may not expect. To combine our intuition with statistical analysis, we will need to visualise complex networks over time and at various levels of abstraction.

This, however, is only half of our task. We must also discover parameterised algorithms that generate the kind of graphs we want. If algorithms can be designed such that small parametric changes generally result in small changes in graphical properties (though not necessarily strict topological changes), we have a mechanism to move through the space of graphs; this will allow structure to be programmed by learning and optimisation. Essentially, we need to know what we want, and how to get it.

Biology is filled with computational systems with graphical properties to study. Biology presents us with a place to start looking for good examples of structure, and, by, working closely with biologists, we aid both their understanding of natural processes and deepen our understanding of computational processes.

But this is just the start – our initial inspiration only. There is no good reason to believe that nature has implemented the best algorithms mathematically possible, and we should not dogmatically ape her solutions. The principled way in which to use biological inspiration is as a guide in uncharted computational territory. By observing, experimenting, simulating, generalising, theorising and implementing, we can begin to develop our understanding of non-classical computation. The common thread behind this research should be the investigation of the graphical structure of these computations. Put simply, we must know how to put our bits together if we are to expect them to work.

References:

[1] Adamatzky A. Computing with waves in chemical media: massively parallel reaction-diffusion processors. *IEICETrans Special Issue on New Systems Paradigms for Integrated Electronics.* (2004),

[2] Harding S. and Miller J. Evolution in Materio: A tone discriminator in liquid crystal. *In processings of the Congress on Evolutionary Computation 2004* (CEC 2004). Volume 2, pages 1800 – 1807.

[3] Kumar, S. and Bentley, P. J. (Contributing Eds.) (2003) *On Growth, Form and Computers*. Academic Press, London.

[4] J. F. Miller, P. Thomson. Cartesian Genetic Programming. *Third European Conference on Genetic Programming Edinburgh, April 15-16, 2000*, Proceedings published as Lecture Notes in Computer Science, Vol. 1802, pp. 121-132 Poli, R., Banzhaf, W., Langdon,W.B., Miller, J. F., Nordin, P., Fogarty, T.C., (Eds.)