Computational Irreducibility and Computational Analogy

Hervé Zwirn
UFR de Physique (LIED, Université Paris 7)
CMLA (ENS Cachan, France)
and
IHPST (CNRS, France)
herve.zwirn@gmail.com

In a previous paper [1], we provided a formal definition for the concept of computational irreducibility (CIR), that is, the fact that for a function f from $\mathbf{N}$ to $\mathbf{N}$ it is impossible to compute $f(n)$ without following approximately the same path as computing successively all the values $f(i)$ from $i = 1$ to n. Our definition is based on the concept of enumerating Turing machines (E-Turing machines) and on the concept of approximation of E-Turing machines, for which we also gave a formal definition. Here, we make these definitions more precise through some modifications intended to improve the robustness of the concept. We then introduce a new concept: the computational analogy, and prove some properties of the functions used. Computational analogy is an equivalence relation that allows partitioning the set of computable functions in classes whose members have the same properties regarding their CIR and their computational complexity.

1. Introduction

The notion of computational irreducibility (CIR) seems to have been first put forward by Wolfram. Given a physical system whose behavior can be calculated by simulating explicitly each step of its evolution, is it always possible to predict the outcome without tracing each step? Is there always a shortcut to go directly to the n^{th} step? Wolfram conjectured [2–4] that in most cases the answer is no. While many computations admit shortcuts that allow them to be performed more rapidly, others cannot be sped up. Computations that cannot be sped up by means of any shortcut are called computationally irreducible.

This question has been widely analyzed in the context of cellular automata by Wolfram [3, 5]. A cellular automaton is computationally irreducible if in order to know the state of the system after n steps there is no way other than to evolve the system n times according to the equations of motion. The intuition behind this definition is that

https://doi.org/10.25088/ComplexSystems.24.2.149

there is no way to reach the n^{th} state other than to go through the $(n-1)$ previous ones. Figures 1 and 2 show the behavior of the linear cellular automaton following rule 110. Figure 1 shows the first 25 steps, while Figure 2 shows a much larger number of steps. The behavior of this automaton seems computationally irreducible.

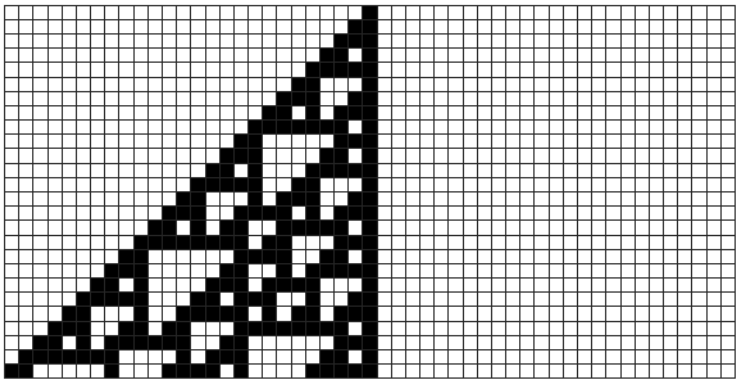

Figure 1. The first 25 steps following rule 110.

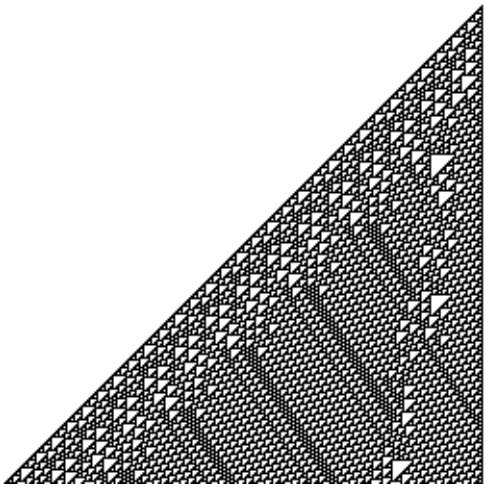

Figure 2. A large number of steps following rule 110.

In this context, Israeli and Goldenfeld [6] have shown that some automata that are apparently computationally irreducible nevertheless have properties that are predictable. But these properties are obtained by coarse graining and do not account for small-scale details. More-

https://doi.org/10.25088/ComplexSystems.24.2.149

over, some automata (e.g., rule 30) seem to be impossible to coarse grain.

Reisinger et al. [7] show that CIR seems to be contingent upon the representation of a given problem. To do so, they consider a game for which the initial rules are computationally reducible, and they build an isomorphic representation leading to a process that appears to be computationally irreducible. As they notice, a more definitive claim would be to take one of Wolfram's computationally irreducible cellular automata, formulate an isomorphic representation of it, and then determine whether transition rules of the equivalent system are computationally reducible.

Whatever the answers to the questions raised by Israeli and Goldenfeld or by Reisinger et al. are, what is of interest for us in this paper is to provide a robust formal definition of the very concept of CIR, which is lacking. Indeed, as we explained in [1], Wolfram's intuition needs to be rigorously formalized, since as stated, it is not robust. There are two underlying intuitions that seem to be equally important in the concept of CIR. The first one is the question of the speed of computation. If a process is computationally irreducible, then it should not be possible to compute its n^{th} state in a time shorter than the time needed to compute successively the $(n-1)$ previous states before computing the n^{th}. The second one is even more demanding. After all, it could well be possible that the time to compute the n^{th} state is not shorter than the sum of the times needed to compute successively all the previous states, but that the computation of the n^{th} state does not really need to go through the computation of these states. But for a process to be computationally irreducible, the necessity to actually compute these previous states is required. Of course, the second condition implies the first one. In the following, we will address both conditions.

In [1], we provided a first formal definition for the concept of CIR, which we reexpressed in the more general framework of functions f from $\mathbf{N}$ to $\mathbf{N}$ as the fact that it is impossible to compute $f(n)$ without following approximately the same path as computing successively all the values $f(i)$ from $i = 1$ to n. Our definition is based on the concept of enumerating Turing machines (E-Turing machines) and on the concept of approximation of E-Turing machines, for which we also gave a formal definition.

In the present paper, we make these definitions more precise and add some modifications intended to improve the robustness of the concept. We refer the reader to the original paper for the motivations of the initial definitions. Here, we also introduce a new concept: the computational analogy.

https://doi.org/10.25088/ComplexSystems.24.2.149

In Section 2, we justify the computation model we use throughout this paper. In Section 3, we make the definition of the E-Turing machines and their approximations precise, and we give more details on the definition of the concept of CIR. In Section 4, we introduce computational analogy, discuss its meaning, and prove some theorems for functions that are computationally analogous, relative to their CIR and their computational complexity.

2. The Computational Model

In this paper, we adopt the computational model of Turing machines [8–11] with $k \geq 2$ tapes. Let us begin by justifying our choice to use the k-tape Turing machines as a good computational model. We are looking for a general model of computation allowing us to deal with the questions of efficiency and speed of computation in a robust way. It is well known that the model of Turing machines is a powerful though very fundamental model of computation. The main point with the Turing machine model is that it is very simple and that through the Church–Turing thesis, it allows the computation of any computable function. Several kinds of Turing machines exist, depending on the number of tapes they have. While they are all equivalent regarding the functions they allow to be computed, they are not equivalent regarding the speed of computation. For example, it is possible to prove that the problem of deciding if a string is a palindrome is $O(n^2)$ in the 1-tape Turing machine model and $O(n)$ in the 2-tape Turing machine model [9, 11]. The first result comes from the fact that it is possible to simultaneously: (1) prove that any 1-tape Turing machine for deciding palindromes must take time (n^2); and (2) exhibit a Turing machine doing the job in $O(n^2)$. The second one comes from the fact that it is possible to exhibit a 2-tape Turing machine deciding a palindrome in $O(n)$, which is obviously the best possible time, since the input of length n has to be read. Does increasing the number of tapes allow us to improve without limit the speed of the computation of a given problem? This answer is no. A first result [11] says that we cannot expect more than a quadratic saving through allowing an arbitrary number of tapes. See Appendix A for the definition of the standard asymptotic notations.

Theorem 1. Given any k-tape Turing machine M operating within time $T(n)$, it is possible to construct a 1-tape Turing machine M' operating within time $O(T(n)^2)$, such that for any input x, $M(x) = M'(x)$.

The meaning of this result is the following: assume that the best 1-tape Turing machine doing a given computation operates in a time

https://doi.org/10.25088/ComplexSystems.24.2.149

$T(n)$. Then, the best k-tape machine that can be designed for doing the same computation will never operate in less than $\sqrt{T(n)}$.

A second result [11] is known as linear speedup.

Theorem 2. For any k-tape Turing machine M operating in time $T(n)$, there exists a k'-tape Turing machine M' operating in time $f'(n) = \epsilon\, T(n) + n$ (where ϵ is an arbitrary small positive constant) that simulates M.

This linear speedup means that the main aspect of complexity is captured through the function $T(n)$ irrespectively of any multiplicative constant. $\mathbf{DTIME}\big(T(n)\big)$ is the class of functions, or class of decision problems, computable by a k-tape Turing machine in $T(n)$ steps. This result means that $\mathbf{DTIME}\big(T(n)\big) = \mathbf{DTIME}\big(\epsilon\, T(n)\big)$, and so it is legitimate to define $\mathbf{DTIME}\big(T(n)\big)$ as the class of functions computable by a Turing machine in $\mathrm{O}\big(T(n)\big)$ steps. If a function f is computable in time $T(n)$ and if $\log\big(f(n)\big)$ (the length of its binary representation) is $\mathrm{o}\big(T(n)\big)$, then f is also computable in time $\epsilon\, T(n)$ for every $\epsilon > 0$.

Hence, in the k-tape Turing machine model, the speed of computation can be expressed through the $\mathrm{O}\big(T(n)\big)$ notation, which is justified. That is what we will do throughout the paper, as is usual in the field of computational complexity.

More results about the so-called "speedup theorems" are given in our previous paper [1].

Usually, in the theory of computation, we are only interested in knowing if a function is computable, and if so, in knowing the computational complexity of getting the output from the input. What is done during the computation is rarely considered, and, except for the person writing the program itself, the Turing machine is a kind of black box furnishing an output from an input. But in this paper, we are interested in a particular aspect of computation that is not often addressed: the intermediate results. As we stated in the introduction, a cellular automaton is computationally irreducible if in order to know the state of the system after n steps there is no way other than to evolve the system n times according to the equations of motion. Similarly, for a function to be computationally irreducible means that the computation of $f(n)$ requires the previous computation of all the $f(i)$ for $i < n$. Computationally irreducible functions are defined not by an explicit formula giving the value of $f(n)$ directly from the value of n, but by recursive rules giving the way to go from $f(i)$ to $f\big(i+1\big)$. Of course, that does not mean that it is enough for a function to be defined by recursive rules to be computationally irreducible. Following these rules, the computation of $f(n)$ starts by the computation of $f\big(1\big)$,

https://doi.org/10.25088/ComplexSystems.24.2.149

followed by the computation of $f(2)$ from $f(1)$, then of $f(3)$ from $f(2)$ and so on, until the computation of $f(n)$ from $f(n-1)$. In order to be able to characterize that sort of computation, our computational model should allow the intermediate computation steps to be identified. For that, we will consider special 3-symbols (0, 1, #) Turing machines such that each of these intermediate results will be successively written on the output tape, with the symbol "#" written at its left. More precisely, a program that follows a recursive rule for computing step by step through the iteration of the same rule "knows" when it switches to the next iteration. What we demand in our specific model of computation is that the intermediate result that is the input of the next iteration be written on the output tape at the right of the symbol "#." The final result will appear on the output tape at the right of the last symbol "#." The output tape will be a one-way tape (i.e., the head will be allowed to go only in the right direction). We will see throughout the paper why this kind of special Turing machine is useful for our purpose.

The goal is to be able to distinguish the different results when reading the output tape. Instead of using a special symbol to separate the results, an equivalent method would be to use a self-delimiting way to write them.

In the following f, g, h, F, G, H will always be functions from $\mathbf{N}$ to $\mathbf{N}$, and M, P, Q will always be Turing machines as described.

3. Computational Irreducibility

Given a Turing machine M computing $f(n)$ in time $T(M(n))$, let us denote by $R_{n,1}, \ldots, R_{n,i}, \ldots, R_{n,T(M(n))}$ the content of the output tape of M during the computation of $f(n)$ after one step of computation, … , i steps of computation, and $T(M(n))$ steps of computation. So $(R_{n,1}, \ldots, R_{n,i}, \ldots, R_{n,T(M(n))})$ is the sequence of the configurations of the output tape during the computation of $f(n)$.

Definition 1 (E-Turing machine). A Turing machine M_f will be called an E-Turing machine for f if:

1. M_f computes f (i.e., for every input n, M_f computes $f(n)$ and halts). It is important to notice that it is the same Turing machine that on input n computes $f(n)$: f is uniformly computed by M_f.
2. During the computation of $f(n)$, there exist increasing $k_n(i)$ for $i = 1$ to $n-1$, such that $f(i)$ is written on the output tape $R_{n,k_n(i)}$ at the right of the last symbol "#."

An E-Turing machine for a function f (in the following we will always denote as M_f such a Turing machine) is a program that, in a cer-

https://doi.org/10.25088/ComplexSystems.24.2.149

tain sense, enumerates the successive values $f(i)$ for $i \leq n$. So, during the computation of $f(n)$, $f(1)$ then $f(2)$, and so on until $f(n)$ successively appear on the output tape of M_f. It is of course possible to build E-Turing machines for any computable function.

Let f be a computable function. Here are two examples of an E-Turing machine for f.

1. Assume first that M is a Turing machine that on every input n computes $f(n)$. Let us now consider the Turing machine M_f, which on every input n calls M with input 1, then, when M has computed $f(1)$, writes "#" and $f(1)$ on the output tape, calls M again with input 2, and so on until the last call to M with input n, and which halts when M has computed $f(n)$, after having written "#" and $f(n)$ on the output tape. M_f is clearly an E-Turing machine for f. When computing $f(n)$, M_f will follow exactly the same initial segments as the initial segments followed for all $k < n$ when computing $f(k)$. The computation of $f(n)$ is the continuation of the computation of $f(k)$ for $k < n$. Also notice that the computation of f for each value n starts from scratch (i.e., the values of $f(k)$ for $k < n$ are not used for computing $f(n)$). This way to build an E-Turing machine is possible for any computable function.
2. Assume now that f is such that it is possible to compute $f(n)$ from $f(n-1)$. Let M' be a Turing machine that on input $f(n-1)$ computes $f(n)$. Let us now consider the Turing machine M'_f, which on every input n starts by computing $f(1)$, writes "#" and $f(1)$ on the output tape, then calls M' to compute $f(2)$ from the input $f(1)$), writes "#" and $f(2)$ on the output tape, and so on until $f(n)$. M'_f is an E-Turing machine for f. The computation of $f(n)$ by M_f can be seen as the successive computations of $f(i)$ from $f(i-1)$ until reaching $f(n)$. As in the first example, when computing $f(n)$, M'_f follows exactly the same initial segments as the initial segments followed for all $k < n$ when computing $f(k)$. Here again, the computation of $f(n)$ is the continuation of the computation of $f(k)$ for $k < n$.

Because the initial path is the same when computing $f(n)$ and $f(m)$ for $n > m$, these two examples of E-Turing machines can be thought of as doing a computation such that on any input n, they halt after having run through an initial segment of length $T(M_f(n))$ of one unique infinite virtual computation of $f(i)$ for $i = 1$ to ∞. That means also that the $k_n(i)$ are independent of n. But this is not necessarily the case for all E-Turing machines.

The computation of $f(n)$ from $f(n-1)$ can be faster than the computation of $f(n)$ from n. In this case, M'_f will be much faster than M_f. We will see that this is the case if f is computationally irreducible, because a Turing machine computing a computationally irreducible function f does need to know $f(n-1)$ (or a value that is near in a sense

https://doi.org/10.25088/ComplexSystems.24.2.149

that we will explain) to compute $f(n)$. We give here some examples of functions that seem intuitively to be more and more "difficult."

In the examples that follow, the arguments given for assessing the "difficulty" of the given function are simply intuitive. We must stress the fact that we do not know how to rule out the possibility that a clever algorithm may be found for speeding up the computation. We just do not know of such an algorithm yet.

- For computing $f(n) = 3^n$ from the input n, a Turing machine will go through some of the intermediate values $f(i)$ for $i < n$ but not necessarily all. For instance, 3^{2n} can be computed as $3^n \times 3^n$, and 3^{2n+3} will need the computation of 3^{n+1} or the computation of 3^n and 3^3. But if $f(n-1)$ is given as input, the computation of $f(n)$ is immediate and fast.
- For computing $f(n) = n!$, a Turing machine will go through n intermediate values if it starts either with n or with $(n-1)!$ as input. Indeed, even from $(n-1)!$ it is necessary to know n for computing $n!$, and a natural way (but not the only one) to "extract" the value n from $(n-1)!$ is to compute all the increasing values of the factorial function and to count how many have been computed until reaching $(n-1)!$. The computation from n can be done in any possible order, since the multiplication of the n first natural numbers can be done from any combination of these numbers. That means that even if a Turing machine computing $n!$ from n will have to perform n operations, it will not necessarily compute all the $k!$ for $k < n$ first. So it seems that every natural Turing machine computing $n!$ with either n or $(n-1)!$ alone as input will have to perform n operations without having to be necessarily an E-Turing machine. But that will not be the case with the input $(n, (n-1)!)$, from which the computation will be very fast. Of course, these considerations are not enough to rule out the possibility of a way to compute $n!$ much more efficiently (which we do not know yet).
- For computing $f(n)$ defined by: "the first bit of the sum of the k^{th} bit of 3^k for all $k \leq n$" from the input n, a Turing machine will go through all the intermediate values $f(i)$ for $i < n$ but will be simply unable to compute $f(n)$ from $f(n-1)$ alone because there is no way to extract the value of n from $f(n-1)$, and this value is needed to compute $f(n)$. So it seems that every Turing machine computing $f(n)$ with n as input will be an E-Turing machine, and $f(n)$ could well be computationally irreducible. From the input $(n, f(n-1))$, the computation will be fast.

The time $T\left(M_f(n)\right)$ to compute $f(n)$ with an E-Turing machine M_f is the sum of the times between the apparition on the output tape of $f(i)$ and $f(i+1)$ (from $i = 1$ to $n-1$) plus the initial time to get $f(1)$ appearing.

Let us denote as $t_i = T\left(f(i-1) \underset{M_f}{\rightarrow} f(i)\right)$ the time between the apparition of $f(i-1)$ and the apparition of $f(i)$ during the computation of $f(n)$ for any $n > i$.

https://doi.org/10.25088/ComplexSystems.24.2.149

We have $T\big(M_f(n)\big) = \sum_{i=1}^{n} t_i$ (we suppose by convention that t_1 is the time for $f(1)$ to appear on the output tape). Since M_f is a Turing machine, t_i is the number of steps done by the machine and so is a strictly positive integer. So $T\big(M_f(n)\big) \geq n$. But in the following, we will be interested only in functions f such that $T\big(M_f(n)\big) = \Omega(n \log n)$.

This seems a reasonable assumption and it is obviously true of any function f such that $f(n) \geq n$, since writing an output n in binary or in any other base ≥ 2 needs at least a time $\log n$, and an E-Turing machine performs n such operations before halting. So the time for an E-Turing machine to compute such a function is necessarily greater than $\sum_{i=1 \text{ to } n} \log i = \log(n!) = \Theta(n \log n)$. So $T\big(M_f(n)\big) = \Omega(n \log n)$.

This is true in particular for the simulation of a large number of nontrivial one-dimensional elementary cellular automata with nearest neighbors (that are $\Omega(n^2)$) and in the majority of the simulations of more complex cellular automata. Of course, we will consider as well computationally irreducible functions for which $f(n) < n$. This is the case of the two candidates given at the end of Section 3, but it is highly probable that they satisfy nonetheless $T\big(M_f(n)\big) = \Omega(n \log n)$.

The question of knowing whether there is an asymptotically optimal program for doing a given computation is a difficult and open question in general. We mean by asymptotically optimal program, a program p such that for any other program p' doing the same computation $T(p(n)) = \mathrm{O}\big(T(p'(n))\big)$. On the one hand, it is well known that the so-called Blum speedup theorem [12] shows that for some decision problems, any program that solves the problem will be much slower than some other program solving the same problem. In these cases, there exists an infinite sequence of programs solving the problem, such that each program in the sequence is much faster than the program it follows, and (up to a multiplicative constant) there is no asymptotically optimal program. But these problems are artificially constructed to prove the theorem. On the other hand, Levin's optimal search theorem [13] proves that for a wide class of problems there is an asymptotically optimal program. These are problems for which verifying a solution is easy, while producing a solution might be difficult. More precisely, these are problems for which the time complexity of checking a solution is asymptotically faster than the time complexity of producing a solution. It is widely thought that no "natural problem" is subject to Blum speedup and that, in general, asymptotically optimal algorithms exist for them. In particular, this is the case for the cellular automata that are the initial source of inspiration for the subject of this paper. Indeed, to show that a program P is asymptotically optimal, it is enough to show that there is a lower bound, say

https://doi.org/10.25088/ComplexSystems.24.2.149

$h(n)$, on the time complexity of any program Q for this problem, $T\big(Q(n)\big) = \Omega\big(h(n)\big)$, and to prove that $T(P(n)) = \mathrm{O}\big(h(n)\big)$. In this case, P is an asymptotically optimal program. For example, in the case of the simulation of nontrivial one-dimensional elementary cellular automata with nearest neighbors, it is clear that any algorithm computing the n initial configurations will have in the worst case to perform in $\Omega\big(n^2\big)$, and that there are algorithms performing in $\mathrm{O}\big(n^2\big)$ (see [1] for details on this point). These algorithms will be asymptotically optimal. Hence, any Turing machine representing these algorithms will be an asymptotically optimal program for the given cellular automaton. This is what we call later an efficient E-Turing machine. We will make the assumption that there always exists an asymptotically optimal Turing machine that we will denote as M_f^* and an efficient E-Turing machine that we will denote as M_f^{eff} for any function f we consider. Put differently, let us say that we restrict our scope to the subset of the computable functions set made of functions that satisfy this requirement (which is hopefully a very large subset).

We give now the formal definition of an efficient E-Turing machine for a function, which will be a fundamental building block for what follows.

Definition 2 (efficient E-Turing machine). We will say that an E-Turing machine M_f^{eff} for f is an efficient E-Turing machine for f if for any other E-Turing machine M_f for f, $T\big(M_f^{\mathrm{eff}}(n)\big) = \mathrm{O}\big(T\big(M_f(n)\big)\big)$; that is, there are constants $c > 0$, $n_0 > 0$ such that $\forall n > n_0$, $T\big(M_f^{\mathrm{eff}}(n)\big) \leq c\, T\big(M_f(n)\big)$.

As explained, the intuition is that asymptotically it is not possible for an E-Turing machine to compute faster than an efficient E-Turing machine.

It is clear from the definition that for any two efficient E-Turing machines M_f^{eff}, $M_f'^{\mathrm{eff}}$, and for any two asymptotically optimal Turing machines M_f^*, $M_f'^*$, we have $T\big(M_f^{\mathrm{eff}}(n)\big) = \Theta\big(T\big(M_f'^{\mathrm{eff}}(n)\big)\big)$ and $T\big(M_f^*(n)\big) = \Theta\big(T\big(M_f'^*(n)\big)\big)$. So for any function H, $H(n) = O\big(T\big(M_f^{\mathrm{eff}}(n)\big)\big)$ is equivalent to $H(n) = \mathrm{O}\big(T\big(M_f'^{\mathrm{eff}}(n)\big)\big)$, and $H(n) = \mathrm{O}\big(T\big(M_f^*(n)\big)\big)$ is equivalent to $H(n) = \mathrm{O}\big(T\big(M_f'^* f(n)\big)\big)$. In the following, M_f^{eff} will always denote an efficient E-Turing machine for f, and $T\big(M_f^{\mathrm{eff}}(n)\big)$ will denote the time for an efficient E-Turing machine to compute $f(n)$. M_f^* will always denote an asymptotically optimal Turing machine computing f, and $T\big(M_f^*(n)\big)$ will denote the time for an asymptotically

https://doi.org/10.25088/ComplexSystems.24.2.149

optimal Turing machine to compute $f(n)$. According to this, there will be no need to be precise about which particular efficient E-Turing machine or which asymptotically optimal Turing machine is considered.

We always suppose that there exist an asymptotically optimal Turing machine M_f^* and an efficient E-Turing machine M_f^{eff} for f.

Definition 3 (approximation of an E-Turing machine). A Turing machine M will be said to be a P-approximation (or simply, approximation) of an E-Turing machine for f if and only if there are a function F such that $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right) \Big/ n\right)$ and a Turing machine P such that for every n:

1. On input n, M computes a result r_n such that P computes $f(n)$ from n and r_n in a number of steps $F(n)$ and halts.
2. During the computation, there exist nondecreasing $k_n(i)$ for $i = 1$ to $n - 1$, such that a result $r'_{n,i}$ is written on the output tape $R_{n,k_n(i)}$ at the right of the last symbol "#," and P computes $f(i)$ from n, i, and $r'_{n,i}$ in a number of steps $F(i)$ and halts. (We will often omit to mention again the inputs n, i, which will be implied.)

Actually, if we note $r_n = r'_{n,n}$, P always computes from the triplet $\left(n, i, r'_{n,i}\right)$, here abbreviated as n, r_n when $i = n$.

Intuitively, an approximation of an E-Turing machine for f is a Turing machine doing a computation that is near the computation made by an E-Turing machine for f.

Let us notice that each E-Turing machine for f is of course an approximation of an E-Turing machine for f. The associated Turing machine P is simply the identity (a Turing machine that computes n from the input n) under the condition that $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right) \Big/ n\right) = \Omega\left(l\left(f(n)\right)\right)$.

An approximation P of an E-Turing machine for f can be an E-Turing machine for r if the $r'_{n,i}$ do not depend on n and if $r'_i = r_i$ for all i (i.e., the intermediate results are the values actually computed by P). But it is not necessarily always the case. In particular, it can happen that the intermediate results $r'_{n,i}$ from which P computes $f(i)$ are different for different values of n. In this case, the path that M follows for computing r_n is different for different values of n, and the r_i for $i < n$ are not necessarily computed.

The concept of approximation of an E-Turing machine for f is actually a concept obtained from the concept of an E-Turing machine by relaxing the constraints of the definition along three dimensions. The first one is the fact that on input n an approximation does not compute exactly $f(n)$ but a value $r(n)$ such that it is possible to go from $r(n)$ to $f(n)$ through a very short computation. The second one is that the intermediate results do not need to be all the $f(i)$ for $i < n$ but

https://doi.org/10.25088/ComplexSystems.24.2.149

values $r_{n,i}$ from which it is possible to compute $f(i)$ through a very short computation, and the third one is that it is not even necessary that the intermediate values be the same on every computation for different n.

Another point to notice is that we do not claim that it is necessary to be able to build the Turing machine P that is associated to an approximation through an effective means. We only ask that such a machine exist.

We can intuitively justify the value chosen for $F(n)$. $F(n)$ is the time that the computation of $f(n)$ takes from the value r_n that is computed by the approximation. We have in mind the case of computationally irreducible functions for which computing $f(n)$ demands computing all the previous values. For these functions, since we want r_n to be "near" $f(n)$ in a certain sense, the time to go from r_n to $f(n)$ must be very short compared to the time to compute $f(n)$ from n, and at most comparable to the time to compute $f(n)$ from $f(n-1)$. That is the reason why $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$. Indeed, if f is computationally irreducible, we will see that this is the average time to compute $f(n)$ from $f(n-1)$. The factor $1/n$ in $T\left(M_f^*(n)\right)/n$ takes into account the fact that there are n necessary phases to compute $f(n)$ with an E-Turing machine for f, and that we want P to compute in a time shorter than or equal to each one of these phases.

Another way to understand the value of $F(n)$, coming from the picture of cellular automata, is to think that r_n is "near" $f(n)$ (and then the computation of $f(n)$ from r_n is fast) if there are only a bounded number of operations to perform on some bits of r_n to go from r_n to $f(n)$. Indeed, in this framework, a bit of $f(n)$ or of r_n is a cell of the cellular automaton. That means that $F(n)$ is $\mathrm{O}\left(l(r_n)\right)$ where $l(r_n)$ is the length of r_n. A reasonable assumption is that the length of r_n should not exceed much the length of $f(n)$, so $l(r_n) = \mathrm{O}\left(\log f(n)\right)$. That means that $F(n)$ is $\mathrm{O}\left(\log f(n)\right)$. Now as we saw before, $T\left(M_f^{\mathrm{eff}}(n)\right) = \Omega\left(n \log f(n)\right)$, so $\log f(n) = \mathrm{O}\left(T\left(M_f^{\mathrm{eff}}(n)\right)/n\right)$; then $F(n)$ must be $\mathrm{O}\left(T\left(M_f^{\mathrm{eff}}(n)\right)/n\right)$. Now for computationally irreducible functions, we anticipate that $T\left(M_f^{\mathrm{eff}}(n)\right) = \Theta\left(T\left(M_f^*(n)\right)\right)$, so $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$ is equivalent to $F(n) = \mathrm{O}\left(T\left(M_f^{\mathrm{eff}}(n)\right)/n\right)$. For functions that are not computationally irreducible but instead satisfy $T\left(M_f^*(n)\right) = \mathrm{o}\left(T\left(M_f^{\mathrm{eff}}(n)\right)\right)$, the value $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$ is the smaller of the two.

Is it possible to be more demanding and to ask that $F(n)$ be smaller than that? The answer is no, as it is easy to see from the example of one-dimensional cellular automata. $F(n)$ is the time for P to compute

https://doi.org/10.25088/ComplexSystems.24.2.149

$f(n)$ from r_n, so in order for P to read r_n and to write $f(n)$, $F(n)$ must be at least equal to $l(f(n))$. For nontrivial automata, $l(f(n)) = \mathrm{O}(n)$. If these automata are computationally irreducible, then $T(M_f^{\mathrm{eff}}(n)) = \Theta(T(M_f^*(n))) = \mathrm{O}(n^2)$, and so $F(n) = \mathrm{O}(T(M_f^*(n)) / n) = \mathrm{O}(n)$. It can be seen that demanding a smaller value for $F(n)$ would result in no machine P being able to exist, since the time to write $f(n)$ is $\Omega(n)$. Notice that for these automata that are not computationally irreducible and for which $T(M_f^*(n)) = \mathrm{o}(n^2)$, there will be no machine P and hence no approximation of an E-Turing machine. Indeed, in this case $F(n) = \mathrm{O}(T(M_f^*(n)) / n) = \mathrm{o}(n)$, which is too small a value for any P to write $f(n)$. Of course, this is true only for functions such that $l(f(n)) > n$, which is not mandatory. In particular, this is false for trivial automata whose configurations vanish after some iterations or for which the successive configurations are restricted to one cell. So the reasoning given is not a proof but only an intuitive justification of the value of $F(n)$.

Definition 4 (computation of *f*(*n*) based on an approximation). Let M be a P-approximation of an E-Turing machine for f. Let us consider the computation of $f(n)$ done initially through M with input n and continued when M has computed r_n by P, which computes $f(n)$ from n and r_n in a time $F(n)$ and halts. This computation will be said to be a computation of $f(n)$ based on the P-approximation M.

Definition 5 (Turing machine computing *f* based on an approximation). Let M be a P-approximation of an E-Turing machine for f, and let us consider the Turing machine M', which, for every n, computes $f(n)$ through a computation based on the P-approximation M. M' will be said to be a Turing machine computing f based on the approximation M.

If M is an E-Turing machine for f, M and M' are identical and M' is of course an E-Turing machine for f. Otherwise, M' is also an approximation of an E-Turing machine for f. The Turing machine P' associated to M' is the same as P; that is, P' computes $f(i)$ from n, i, and $r'_{n,i}$ in a number of steps $F(i)$, except that for the computation on input n, n, and $r'_{n,n}$, P' is the identity, while P computes $f(n)$.

As shown in Theorem 3, the important point is that it is possible to build an E-Turing machine for f from any approximation of an E-Turing machine for f.

Theorem 3. From any M approximation of an E-Turing machine for f, it is possible to build an E-Turing machine M' for f (we will call it the daughter of M), computing in a time $T(M'(n)) = \Theta(T(M(n)))$.

https://doi.org/10.25088/ComplexSystems.24.2.149

Proof. Since M is an approximation of an E-Turing machine for f, there are a Turing machine P and a function F associated, as mentioned in Definition 3. Let us consider the Turing machine built according to the following way: on input n, M' does exactly the same computation as M, but for each $i < n$, after having computed $r_{n,i}$, M' computes $f(i)$ through P with input n, i, $r_{n,i}$ in a time $F(i)$, writes "#" and $f(i)$ on its output tape, resumes the computation, and finally, computes $f(n)$ from n and r_n. It is clear that M' is an E-Turing machine for f. M' computes in a time:

$$T(M'(n)) = T(M(n)) + \sum_{i=1}^{n} F(i) + \mathrm{O}(1) = T(M(n)) + \sum_{i=1}^{n} \mathrm{O}\left(\frac{T\left(M_f^*(i)\right)}{i}\right) + \mathrm{O}(1).$$

Now it is possible to compute $f(n)$ by M followed by P (i.e., a computation of f based on the approximation M) so:

$$T\left(M_f^*(n)\right) = \mathrm{O}\left(T(M(n)) + F(n)\right)$$

$$T\left(M_f^*(n)\right) = \mathrm{O}\left[T(M(n)) + \mathrm{O}\left(\frac{T\left(M_f^*(n)\right)}{n}\right)\right].$$

Hence

$$T\left(M_f^*(n)\right) = \mathrm{O}\left(T(M(n))\right).$$

Then

$$T(M'(n)) = T(M(n)) + \sum_{i=1}^{n} \mathrm{O}\left(\frac{T(M(i))}{i}\right).$$

Now $\sum_{i=1}^{n} F(i)/i = \mathrm{O}(F(n))$ if F is a convex function and $F(n) = \Omega(\log n)$ (see Appendix B). Since any function $\mathrm{O}\left(T(M(i))\right)$ is a convex function $\Omega(\log n)$, we have

$$T(M'(n)) = T(M(n)) + \mathrm{O}\left(T(M(n))\right) = \mathrm{O}\left(T(M(n))\right).$$

As $T(M(n)) < T(M'(n))$, we get $T(M(n)) = \Theta\left(T(M'(n))\right)$. □

We will denote as $\otimes$ this particular form of composition of the two Turing machines M and P. So $M' = P \otimes M$. The composition $\otimes$ is defined for a pair (P, M) when the second argument is an approximation of an E-Turing machine for a given function f and the first one is the associated Turing machine computing $f(i)$ from the intermediate results of M. Of course, this composition is not to be confused with the usual composition $P \circ M$, which runs first the program M and then the program P, with the result of the computation of M as input. An

https://doi.org/10.25088/ComplexSystems.24.2.149

important difference is the computation time. The computation time of $P \circ M$ is the sum of the respective computation times:

$$T((P \circ M)(n)) = T\big(P(\text{output of } M(n))\big) + T(M(n)),$$

while the computation time of $P \otimes M$ is

$$T((P \otimes M)(n)) = \sum_{i=1}^{n} T(P(r_{n,i})) + T(M(n)) + \mathrm{O}(1) = \sum_{i=1}^{n} F(i) + T(M(n)) + \mathrm{O}(1).$$

Theorem 4. No approximation of an E-Turing machine for f can compute faster than an efficient E-Turing machine for f. More precisely, if M is an approximation of an E-Turing machine for f, then $T\big(M_f^{\text{eff}}(n)\big) = \mathrm{O}\big(T(M(n))\big)$.

Proof. Let M' be the daughter of M. Since M' is an E-Turing machine for f, $T\big(M_f^{\text{eff}}(n)\big) = \mathrm{O}\big(T(M'(n))\big)$. By Theorem 3 we have $T(M'(n)) = \Theta\big(T(M(n))\big)$. So $T\big(M_f^{\text{eff}}(n)\big) = \mathrm{O}\big(T(M(n))\big)$. □

Theorem 5. Let M' be a Turing machine computing f based on an approximation M. Then $T(M'(n)) = \Theta\big(T(M(n))\big)$.

Proof. M' will compute in a time $T(M'(n))$ such that

$$T(M(n)) \le T(M'(n)) \le T(M(n)) + F(n) = T(M(n)) + \mathrm{O}\left(\frac{T\big(M_f^*(n)\big)}{n}\right) = T(M(n)) + \mathrm{O}\left(\frac{T(M(n))}{n}\right) = \mathrm{O}\big(T(M(n))\big).$$

So $T(M(n)) = \Theta\big(T(M'(n))\big)$. □

In summary, we can say that an approximation of an E-Turing machine for f, its daughter, and any Turing machine computing f based on this approximation all compute in the same time.

Definition 6 (strong CIR (resp. simple CIR) functions). A function $f(n)$ from $\mathbf{N}$ to $\mathbf{N}$ will be said to have strong CIR (resp. simple CIR) if and only if for any Turing machine M computing f there is a P-approximation of an E-Turing machine for f, M' such that for every n (resp. for infinitely many n), the computation of $f(n)$ by M is based on M'.

The intuition is that if a function is strongly computationally irreducible, for each n there is no way to compute $f(n)$ other than to first compute all the values $f(i)$ for $i < n$ (or values that are near in the sense given in Definition 3). There is no shortcut to directly get the

https://doi.org/10.25088/ComplexSystems.24.2.149

value of $f(n)$ without having first computed $f(n-1)$ or a value that is near $f(n-1)$ and so forth for the previous values. If a function is computationally irreducible (but not strongly computationally irreducible), for infinitely many n there is no way to compute $f(n)$ other than to first compute all the values $f(i)$ for $i < n$ (or values that are near).

The reason it is useful to introduce this distinction between strong CIR and simple CIR can be explained through the following example. Assume that f is strongly computationally irreducible. So there is no way to compute $f(n)$ other than to first compute all the values $f(i)$ for $i < n$ (or values that are near), and that is true for every n. Let us now consider the function g such that $g(2i-1) = f(i)$ and $g(2i) = 1$. It is clear that computing g for any even value is very easy and does not imply having to compute any other result first. So g is not strongly computationally irreducible. But the intuition is nevertheless that g is irreducible in some way. The notion of strong CIR needs to be weakened to cover functions like g and many others that similarly need infinitely often (but not always) to go through the computation of all the previous values in order to be computed. It is worth noticing that if a function is computationally irreducible but not does not have strong CIR, then the computation of $f(n)$ will require fewer than n steps. In the example of the function g, the computation made by any Turing machine computing g can be based on a Turing machine that computes $g(2i-1)$ in i steps through a P-approximation of an E-Turing machine for f and $g(2i)$ in one step (since it is equal to 1). So the computation of g will require at most $n/2$ steps. This example can of course be extended to any other value of the required number of steps, as long as this number is a growing unbounded function of n.

Theorem 6. If a function f has strong CIR, then no Turing machine computing f can compute $f(n)$ faster than an efficient E-Turing machine for f. So for any Turing machine M computing f, $T(M_f^{\text{eff}}(n)) = O(T(M(n)))$.

Proof. If f has strong CIR, then any Turing machine M computing f is based on an approximation of an E-Turing machine for f. Let M' be this approximation. From Theorem 4, $T(M_f^{\text{eff}}(n)) = O(T(M'(n)))$. From Theorem 5, $T(M(n)) = \Theta(T(M'(n)))$.

So $T(M_f^{\text{eff}}(n)) = O(T(M(n)))$. □

This result is slightly weakened in Theorem 7 if f is simply computationally irreducible. In this case, for any Turing machine computing

https://doi.org/10.25088/ComplexSystems.24.2.149

f, there are infinitely many values of $f(n)$ that it is not possible to compute faster than the computation by an efficient E-Turing machine for f.

Theorem 7. If a function f is computationally irreducible, then for any Turing machine M computing f there are constants $c > 0$, $n_0 > 0$ such that $\forall\, N > n_0$, $\exists\, n > N$, $T\left(M_f^{\text{eff}}(n)\right) \leq c\, T(M(n))$.

Proof. If f is computationally irreducible, then for any Turing machine M computing f there is a P-approximation of an E-Turing machine for f, M' such that for infinitely many n, the computation of $f(n)$ by M is based on M'. From Theorem 4, $T\left(M_f^{\text{eff}}(n)\right) = \mathrm{O}\left(T(M'(n))\right)$. So there are constants $c > 0$, $n_0 > 0$ such that $\forall\, n > n_0$, $T\left(M_f^{\text{eff}}(n)\right) \leq c\, T(M'(n))$. But $\forall\, N$, $\exists\, n > N$ such that the computation of $f(n)$ by M is based on M'. For such n, $T\left(M(n) > T(M'(n)\right)$. So for those n that are superior to n_0, $T\left(M_f^{\text{eff}}(n)\right) \leq c\, T(M(n))$. □

Theorem 8. If a function f has strong CIR, then $T\left(M_f^*(n)\right) = \Theta\left(T\left(M_f^{\text{eff}}(n)\right)\right)$.

Proof. If f has strong CIR, then by Theorem 6 for any Turing machine M computing f, $T\left(M_f^{\text{eff}}(n)\right) = \mathrm{O}\left(T(M(n))\right)$. So $T\left(M_f^{\text{eff}}(n)\right) = \mathrm{O}\left(T\left(M_f^*(n)\right)\right)$. Because of the definition of an asymptotically optimal Turing machine, for any Turing machine M computing f, $T\left(M_f^*(n)\right) = \mathrm{O}\left(T(M(n))\right)$. So $T\left(M_f^*(n)\right) = \mathrm{O}\left(T\left(M_f^{\text{eff}}(n)\right)\right)$.

Hence $T\left(M_f^*(n)\right) = \Theta\left(T\left(M_f^{\text{eff}}(n)\right)\right)$. □

Definition 6 and Theorems 6, 7, and 8 address the two key points of the underlying intuitions for the concept of CIR: the speed of computation and the path followed during the computation.

Example 1. Let $\mathcal{B} = \{0, 1\}$ and $\mathcal{B}^*$ be the set of all finite strings over $\mathcal{B}$. Let $\mathcal{L}$ be a recursive language and assume an enumeration of the words of $\mathcal{B}^*$ (e.g., the index in the length-increasing lexicographic ordering). Define the function f by $f(n)$ as the number of words w_i (for $i \leq n$ in the chosen enumeration) of $\mathcal{B}^*$ in $\mathcal{L}$. Then it seems that, in general, there is no other way to compute $f(n)$ than to decide for each $i \leq n$ if the word w_i belongs or not to $\mathcal{L}$ and to count the number of positive answers.

Example 2. Knowing if an initial configuration of Conway's Game of Life will be eternal or not is an undecidable problem. So let $f(n)$ be the number of initial configurations with an index smaller than $n + 1$

https://doi.org/10.25088/ComplexSystems.24.2.149

in a given enumeration that are still living after n iterations. Here again, it seems that there is no way to compute $f(n)$ other than to test each one of the relevant configurations during n steps and therefore, by so doing, to go through the computation of all the $f(i)$ for $i < n$.

4. Computational Analogy

Computational analogy should not be confused with Wolfram's principle of computational equivalence, which states that most systems found in the natural world are computationally equivalent because most of them can perform universal computations. Computational analogy concerns functions that are not necessarily universal but that share properties about their computational complexity (their asymptotically optimal programs compute in the same time as well as their efficient E-Turing machines) and their computational irreducibility (see the comment after Theorem 13).

Let M be an approximation of an E-Turing machine for f. M computes a function r but is not necessarily an E-Turing machine for r. Nevertheless, it is clear that each E-Turing machine for r is an approximation of an E-Turing machine for f. But it is possible that no E-Turing machine for f is an approximation of an E-Turing machine for r. Such would be the case if, while the time to go from n, $r(n)$ to $f(n)$ through P is $O\left(T\left(M_f^*(n)\right) / n\right)$, there is no Turing machine able to compute $r(n)$ from n, $f(n)$ in a time $O\left(T(M_r^*(n)) / n\right)$, where M_r^* is an asymptotically optimal Turing machine for r. But if one E-Turing machine for f is an approximation of an E-Turing machine for r, then every E-Turing machine for f will be an approximation of an E-Turing machine for r. In this case, each E-Turing machine for f is an approximation of an E-Turing machine for r and vice versa, each E-Turing machine for r is an approximation of an E-Turing machine for f. So it is possible to define a relation of "computational analogy" **CA** (which will be proved to be an equivalence relation).

Definition 7 (computational analogy). f and g will be said to be computationally analogous (noted f **CA** g) if:

1. There exists a Turing machine M that is both an E-Turing machine for f and an approximation of an E-Turing machine for g.
2. There exists a Turing machine M' that is both an E-Turing machine for g and an approximation of an E-Turing machine for f.

That means that there is a Turing machine $P^{f \to g}$ that computes $g(n)$ from n, $f(n)$ for every n in a time $F(n) = O\left(T\left(M_g^*(n) / n\right)\right)$ (and vice versa). So we have the following theorems and proofs.

https://doi.org/10.25088/ComplexSystems.24.2.149

Theorem 9. $(f \mathbf{CA}\, g)$ is equivalent to: there is a Turing machine $P^{f\to g}$ that computes $g(n)$ from n, $f(n)$ for every n in a time $F(n) = \mathrm{O}\left(T\left(M_g^*(n)\,/\,n\right)\right)$, and there is a Turing machine $P^{g\to f}$ that computes $f(n)$ from n, $g(n)$ for every n in a time $G(n) = \mathrm{O}\left(T\left(M_f^*(n)\,/\,n\right)\right)$.

In the following, when $f \mathbf{CA}\, g$, we will always denote by $P^{g\to f}$ and $P^{f\to g}$ these Turing machines.

Theorem 10. Let M_f^* (resp. M_g^*) be an asymptotically optimal Turing machine computing f (resp. g). If $f \mathbf{CA}\, g$, then $T\left(M_f^*(n)\right) = \Theta\left(T\left(M_g^*(n)\right)\right)$.

Proof. The Turing machine $\left(P^{f\to g} \circ M_f^*\right)$ computes g in a time $T\left(M_f^*(n)\right) + F(n)$ with $F(n) = \mathrm{O}\left(T\left(M_g^*(n)\,/\,n\right)\right)$. Since M_g^* is an asymptotically optimal Turing machine computing g, $T\left(M_g^*(n)\right) = \mathrm{O}\left(T\left(M_f^*(n)\right)\right) + \mathrm{O}\left(T\left(M_g^*(n)\right)\,/\,n\right)$, so $T\left(M_g^*(n)\right) = \mathrm{O}\left(T\left(M_f^*(n)\right)\right)$. The same reasoning with $\left(P^{g\to f} \circ M_g^*\right)$ proves that $T\left(M_f^*(n)\right) = \mathrm{O}\left(T\left(M_g^*(n)\right)\right)$. Then $T\left(M_f^*(n)\right) = \Theta\left(T\left(M_g^*(n)\right)\right)$. □

Theorem 11. Let M_f^{eff} (resp. M_g^{eff}) be an asymptotically optimal Turing machine computing f (resp. g). If $f \mathbf{CA}\, g$, then $T\left(M_f^{\mathrm{eff}}(n)\right) = \Theta\left(T\left(M_g^{\mathrm{eff}}(n)\right)\right)$.

Proof. The Turing machine $P^{f\to g} \otimes M_f^{\mathrm{eff}}$, which is an E-Turing machine for g, computes in a time

$$T\left(P^{f\to g} \otimes M_f^{\mathrm{eff}}(n)\right) = T\left(M_f^{\mathrm{eff}}(n)\right) + \sum_{i=1}^{n} \mathrm{O}\left(\frac{T\left(M_g^*(i)\right)}{i}\right).$$

Since M_g^{eff} is an efficient E-Turing machine, $T\left(M_g^{\mathrm{eff}}(n)\right) = \mathrm{O}\left(T\left(P^{f\to g} \otimes M_f^{\mathrm{eff}}(n)\right)\right)$.

Hence

$$T\left(M_g^{\mathrm{eff}}(n)\right) = \mathrm{O}\left(T\left(M_f^{\mathrm{eff}}(n)\right) + \sum_{i=1}^{n} \mathrm{O}\left(\frac{T\left(M_g^*(i)\right)}{i}\right)\right).$$

Now

$$\sum_{i=1}^{n} \mathrm{O}\left(\frac{T\left(M_g^*(i)\right)}{i}\right) = \mathrm{O}\left(T\left(M_g^*(n)\right)\right)$$

(see Appendix B).

https://doi.org/10.25088/ComplexSystems.24.2.149

So $T\left(M_g^{\text{eff}}(n)\right) = \mathrm{O}\left(T\left(M_f^{\text{eff}}(n)\right)\right) + \mathrm{O}\left(T\left(M_g^*(n)\right)\right)$. Now $T\left(M_g^*(n)\right) = \Theta\left(T\left(M_f^*(n)\right)\right)$ by Theorem 10, and $T\left(M_f^*(n)\right) = \mathrm{O}\left(T\left(M_f^{\text{eff}}(n)\right)\right)$; then $T\left(M_g^{\text{eff}}(n)\right) = \mathrm{O}\left(T\left(M_f^{\text{eff}}(n)\right)\right)$.

The same reasoning for f shows that $T\left(M_f^{\text{eff}}(n)\right) = \mathrm{O}\left(T\left(M_g^{\text{eff}}(n)\right)\right)$.

Hence $T\left(M_f^{\text{eff}}(n)\right) = \Theta\left(T\left(M_g^{\text{eff}}(n)\right)\right)$. □

Theorem 12. $\left(f\ \mathbf{CA}\ g\right)$ is equivalent to: any approximation of an E-Turing machine for f is an approximation of an E-Turing machine for g and vice versa.

Proof. Consider first the direct sense: Let M be a P-approximation of an E-Turing machine for f. P computes in a time $\mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$. According to Theorem 9, there is a Turing machine $P^{f\to g}$ that computes $g(n)$ from $f(n)$ for every n in a time $F(n) = \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$.

It is then clear that M is a $\left(P^{f\to g}\circ P\right)$-approximation of an E-Turing machine for g because $\left(P^{f\to g}\circ P\right)$ computes in a time $\mathrm{O}\left(T\left(M_f^*(n)\right)/n\right) + \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right) = \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$, since by Theorem 10, $T\left(M_f^*(n)\right) = \Theta\left(T\left(M_g^*(n)\right)\right)$. Consider now the reverse sense: an E-Turing machine for f is an approximation of an E-Turing machine for f, so it is an approximation of an E-Turing machine for g (and vice versa). □

The very meaning of $f\ \mathbf{CA}\ g$ is that f and g share the same approximations of E-Turing machines.

Theorem 13. **CA** is an equivalence relation.

Proof. This is obvious by Theorem 12. □

The quotient set of the computable functions set by this equivalence relation is made of equivalence classes of computationally analogous functions that share properties about their computational complexity (their asymptotically optimal programs compute in the same time as well as their efficient E-Turing machines, by Theorems 10 and 11) and their CIR, as we are now going to show.

Let us recall that we restrict our scope to the computable functions that satisfy the requirement that there be an asymptotically optimal program and an efficient E-Turing machine for them.

Theorem 14. Assume $f\ \mathbf{CA}\ g$. If f has strong CIR, then g also has strong CIR.

Proof. Let M be a Turing machine computing every $g(n)$. Since $f\ \mathbf{CA}\ g$, there is a Turing machine $P^{g\to f}$ that computes $f(n)$ from n,

https://doi.org/10.25088/ComplexSystems.24.2.149

$g(n)$ for every n in a time $F(n) = \mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$. $\left(P^{g\to f}\circ M\right)$ is a Turing machine computing every $f(n)$. Now f has strong CIR, so there are a Turing machine S that is an approximation of an E-Turing machine for f, a Turing machine Q, and a function $H(n) = \mathrm{O}\left(T\left(M_f^*(n)\right)/n\right)$ such that for every n, the computation of $f(n)$ made by $\left(P^{g\to f}\circ M\right)$ is based on S (i.e., is actually the same as the computation of $f(n)$ made by S followed by Q that computes in a time $H(n)$). Since f **CA** g, by Theorem 12, S is also an approximation of an E-Turing machine for g. So during the computation of $f(n)$ there is data $r_{n,i}$ (computed by S) appearing successively in an increasing order from $i = 1$ to n on the output string of S, such that there is a Turing machine Q' that on input $r_{n,i}$, computes $g(i)$ in a number of steps $H'(i)$ (where $H'(n) = \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$). Since $\left(P^{g\to f}\circ M\right)$ and $\left(Q\circ S\right)$ are the same Turing machine, that means that some of these $r_{n,i}$ appear during the computation of M and some appear during the computation of $P^{g\to f}$. Let us assume that all the $r_{n,i}$ for $i = 1$ to k appear during the computation of M and that all the $r_{n,i}$ for $i = k + 1$ to n appear during the computation of $P^{g\to f}$. Let us now consider the Turing machine Q'' gotten from Q' through the following change:

- On input n, i, $r_{n,i}$ for $i = 1$ to k, Q'' does the same computation as Q' (i.e., computes $g(i)$ in a time $H'(i)$).
- On input n, i, $r_{n,k}$ for $i = k + 1$ to n, Q'' starts by computing $r_{n,i}$, then computes $g(i)$ from $r(i)$ as Q' does.

Since $P^{g\to f}$ computes $f(n)$ from n, $g(n)$ in a time $G(n) = \mathrm{O}\left(T\left(M_f^*(n)/n\right)\right)$, all the $r_{n,i}$ for $i = k + 1$ to n will appear in a time less than $G(n)$. So the computation of $g(i)$ from n, i, $r_{n,k}$ (for $i = k + 1$ to n) will be done in a time $H''(i)$ smaller than $G(n) + H'(i)$. Since $G(n) = \mathrm{O}\left(T\left(M_f^*(n)/n\right)\right)$, which is equal to $\mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$ by Theorem 10, and since $H'(n) = \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$, we get $H''(n) = \mathrm{O}\left(T\left(M_g^*(n)\right)/n\right)$.

Let us notice that the list of intermediate results $r'_{n,i}$ from which Q'' computes $g(i)$ is the same as the list of $r_{n,i}$ for $i = 1$ to k and is equal to $r_{n,k}$ for $i = k$ to n. That means that M is based on a Q''-approximation of an E-Turing machine for g (the Turing machine computing all the $r_{n,i}$ for $i = 1$ to k), and so g has strong CIR. □

Theorem 15. Assume f **CA** g. If f has simple CIR, then g also has simple CIR.

https://doi.org/10.25088/ComplexSystems.24.2.149

Proof. Let M be a Turing machine computing every $g(n)$. Since f **CA** g, there is a Turing machine $P^{g\to f}$ that computes $f(n)$ from n, $g(n)$ for every n in a time $F(n) = \mathrm{O}\left(T\left(M_f^*(n)/n\right)\right)$. $\left(P^{g\to f}\circ M\right)$ is a Turing machine computing every $f(n)$. Now f is computationally irreducible, so there is an approximation S of an E-Turing machine for f such that for infinitely many n, the computation of $f(n)$ by $\left(P^{g\to f}\circ M\right)$ is based on S. Let us consider the function f' obtained from f by $f'(n) = f(p)$, where p is the n^{th} value for which the computation of f by $\left(P^{g\to f}\circ M\right)$ is based on S. It is clear that f' has strong CIR, since for every n, the computation of $f'(n)$ is based on the approximation S', which does exactly the same computation as S, except that on input n, S' computes the result that S computes on input p, where p is the n^{th} value for which the computation of f by $\left(P^{g\to f}\circ M\right)$ is based on S. Let g' be the function defined similarly from g: $g'(n) = g(p)$, where p is the n^{th} value for which the computation of f by $\left(P^{g\to f}\circ M\right)$ is based on S. It is clear that f' **CA** g'. So g' has strong CIR. Then g is computationally irreducible. □

5. Conclusion

We have provided a formal definition of computational irreducibility (CIR) that clarifies the intuition about this concept and that allows us to understand that a function is computationally irreducible if there is a class of close paths that it is necessary to follow in order to compute it. In a broad sense, that means that if a function is computationally irreducible, there is only one road (the width of the road being the size of the class of the close paths that can be used) to compute this function. In some sense, all these paths have the same length. This explains the fact that it is not possible to go faster than following one of these paths to compute the function. We have also defined an equivalence relation between functions that share the same road. Roughly speaking, computational analogy allows us to get a quotient set that can be viewed as a map of the computable functions set (or at least a large subset of this set whose elements satisfy the conditions for the concepts discussed to be applicable) for which classes are grouping elements having similar properties relative to their time of computation and their CIR.

An open problem is still to prove that one function among the possible candidates is really computationally irreducible. The cellular automaton rule 110, which has been shown to be universal (see [14]), the third of the three functions we mention at the beginning of Sec-

https://doi.org/10.25088/ComplexSystems.24.2.149

tion 3, or the two other examples we proposed at the end of Section 3 are good examples of functions that we would like to prove computationally irreducible. Proving that a given function f is computationally irreducible amounts to proving that there is no way to compute f except through an algorithm that approximately follows the path of an enumerating Turing machine (E-Turing machine) computing f. Ruling out all the other possible ways to compute f would possibly face the halting problem, hence resulting in the proposition "f is computationally irreducible" being undecidable (I would like to thank one anonymous referee for this remark). But this remains to be proven. An even more difficult task would be to prove that CIR is an undecidable property. In this case, that would not mean that it is always impossible to prove that a given function is computationally irreducible, but that there is no general algorithm for deciding for any function if it is computationally irreducible or not, even if it is possible to prove that some particular functions are computationally irreducible. These difficult points are left for further investigation.

From a more philosophical point of view, CIR can help clarify the concept of emergence and can be used to understand why certain phenomena appear to be emergent. We have proposed in [15] and [16] that "understanding" a process implies having a mental model of it that we can use to simulate its behavior. Emergent phenomena are effects or properties appearing at the macro level (collective) of a system and that are caused by the micro level (individual) but are very difficult and even seemingly impossible to predict, even from the complete knowledge of the rules of the micro level. If the process running at the micro level is computationally irreducible or if the rules leading from the micro level to the macro level are computationally irreducible, then the global behavior of the system will be neither predictable (without simulating it) nor understandable. In this case, what happens will be seen as "emergent." For example, the fact that some patterns (e.g., pulsar, glider, glider gun, etc.) are usually considered as emergent in Conway's Game of Life could be explained by the fact that the underlying rules are computationally irreducible. Similarly, phenomena that are sometimes interpreted as downward causation could be merely computationally irreducible processes interpreted as causal effects between the two levels of description. That is a point that we will address at more length in a forthcoming paper.

Acknowledgment

I am indebted to Jean-Paul Delahaye for many enlightening discussions from the beginning of this work, to Serge Grigorieff for having read a first version of this paper and given a counterexample that led

https://doi.org/10.25088/ComplexSystems.24.2.149

me to modify the initial definition of an approximation and to build the present one, and to Jean-Michel Ghidaglia for a useful discussion on Appendix B.

Appendix

A. The Asymptotic Notations

The asymptotic notations are useful for comparing the order of magnitude of different functions. We recall here the standard notations.

- $f(n) = \mathrm{O}(g(n))$ if there are constants $c > 0$, $n_0 > 0$ such that $\forall\, n > n_0$, $|f(n)| \leq c\,|g(n)|$.
- $f(n) = \mathrm{o}(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.
- $f(n) = \Omega(g(n))$ if there are constants $c > 0$, $n_0 > 0$ such that $\forall\, n > n_0$, $|f(n)| \geq c\,|g(n)|$.
- $f(n) = \omega(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = \infty$.
- $f(n) \sim g(n)$ if $\lim_{n\to\infty} f(n)/g(n) = 1$.
- $f(n) = \Theta(g(n))$ if there are constants $c > 0$, $c' > 0$, $n_0 > 0$ such that $\forall\, n > n_0$, $cg(n) \leq f(n) \leq c'\,g(n)$).

B. A Property of Convex Functions

We prove here that $\sum_{i=1}^{n} F(i))/i = \mathrm{O}(F(n))$ if F is a convex function and $F(n) = \Omega(\log n)$:

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{F(i)}{i} = \lim_{n\to\infty} \int_1^n \frac{F(x)}{x}\,\mathrm{dx},$$

$$F(y) = \int_1^y F'(x)\,\mathrm{dx} + F(1) \text{ for } y > 1,$$

Now if F is convex, $\int_1^y F'(x)\,\mathrm{dx} \leq (y-1)\,F'(y) \leq y\,F'(y)$, so

$$\frac{F(y)}{y} \leq F'(y) + \frac{F(1)}{y},$$

$$\int_1^x \frac{F(y)}{y}\,\mathrm{dy} \leq F(x) - F(1) + F(1)\log x.$$

https://doi.org/10.25088/ComplexSystems.24.2.149

Now if $\log x = \mathrm{O}(F(x))$, then

$$\int_1^x \frac{F(y)}{y}\,\mathrm{d}y = \mathrm{O}(F(x)).$$

References

[1] H. Zwirn and J. P. Delahaye, "Unpredictability and Computational Irreducibility," in *Irreducibility and Computational Equivalence: Wolfram Science 10 Years after the Publication of A New Kind of Science* (H. Zenil, ed.), New York: Springer, 2013 pp. 273–295.

[2] S. Wolfram, "Undecidability and Intractability in Theoretical Physics," *Physical Review Letters*, **54**(8), 1985 pp. 735–738.

[3] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[4] H. Zenil, F. Soler-Toscano, and J. J. Joosten, "Empirical Encounters with Computational Irreducibility and Unpredictability." arxiv.org/abs/1104.3421.

[5] S. Wolfram, "Statistical Mechanics of Cellular Automata," *Reviews of Modern Physics*, **55**(3), 1983 pp. 601–644.

[6] N. Israeli and N. Goldenfeld, "Computational Irreducibility and the Predictability of Complex Physical Systems," *Physical Review Letters*, **92**(7), 2004 p. 074105. doi:10.1103/physrevlett.92.074105.

[7] D. Reisinger et al., "Exploring Wolfram's Notion of Computational Irreducibility with a Two-Dimensional Cellular Automaton," in *Irreducibility and Computational Equivalence: Wolfram Science 10 Years after the Publication of A New Kind of Science* (H. Zenil, ed.) New York: Springer, 2013 pp. 263–272.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability*, San Francisco: W. H. Freeman, 1979.

[9] O. Goldreich, *Computational Complexity: A Conceptual Perspective*, New York: Cambridge University Press, 2008.

[10] R. H. Hartley, *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill, 1967.

[11] C. H. Papadimitriou, *Computational Complexity*, Reading, MA: Addison-Wesley, 1994.

[12] M. Blum, "A Machine-Independent Theory of the Complexity of Recursive Functions," *Journal of the ACM*, **14**(2), 1967 pp. 323–336. doi:10.1145/321386.321395.

[13] L. A. Levin, "Universal Sequential Search Problems," *Problems of Information Transmission*, **9**(3), 1973 pp. 265–266.

https://doi.org/10.25088/ComplexSystems.24.2.149

[14] M. Cook, "Universality in Elementary Cellular Automata," *Complex Systems*, **15**(1), 2004 pp. 1–40.
http://www.complex-systems.com/pdf/15-1-1.pdf.

[15] H. Zwirn, *Les limites de la connaissance*, Paris: Odile Jacob, 2000.

[16] H. Zwirn, *Les systèmes complexes*, Paris: Odile Jacob, 2006.

https://doi.org/10.25088/ComplexSystems.24.2.149