A Simplified Calculation of Integrated Information based on Formal and Numerical Connections with Algorithmic Information

Alberto Hernández-Espinosa, Hector Zenil

Abstract

In previous approaches it has been suggested that algorithmic complexity is somehow connected to integrated information. Here we tackle the suggestion in a formal and systematic fashion and we introduce a novel method to calculate a ϕ based on these connections and a perturbation analysis test instead of rather disconnected steps that had been made in the past assuming that such connections existed. The calculation is based on the idea that simple rules can simulate behaviour. From the perturbation test we demonstrate how we can deduce key causal properties from a system with a significantly reduced number of calculations. This helps avoid one of the limitations of bottleneck principle from other approaches to ϕ. Analysis of information obtained from the perturbation test made possible to find that information distribution in systems here analyzed obey a fractal distribution that can be expressed as a simple, highly compressible, and thus low Kolmogorov complexity model allowing a more general implementation of an automatic perturbation test where a system is 'interrogated' for sets of simple rules, not only for giving an account of its computational capabilities, but also for explanations of its own behaviour.

1 Integrated Information Theory (IIT)

We introduce ϕ_δ, a measure inspired by Tononi's ϕ to compute integrated information, but in contrast to more 'traditional' approaches to ϕ, ϕ_δ is calculated from a perturbation analysis and its connections to algorithmic complexity that we explore both formally and numerically.

Oizumi and Tononi et. al. developed a theory [6] and a toolbox [5] to measure the amount and type of conscious experience using the ϕ parameter.

Among the most salient properties that identify IIT are:

1. "[...] at the fundamental level, consciousness is integrated information, and that its quality is given by the informational relationships generated by a complex of elements"[8].

2. "IIT essentially proposes that quantitative consciousness, i.e. the degree to which a physical system is conscious, is identical to its state-dependent level of maximally integrated conceptual information, which can be quantified by a measure called 'ϕ-max' [3].

As expected, IIT has a set of concepts and methodologies to measure conscious experience, this theoretical framework is well explained in [6].

ϕ_δ follows strictly IIT theoretical framework, but, in contrast to ϕ's IIT, that computes information integration using advanced statistical and mathematical methodologies, ϕ_δ is part of what we called natural computationalism, that, in turn, is based on natural computation paradigm.

We will simply call ϕ the measure proposed over the last years by the Tononi group and others in its different versions that require the full computation of what is called the input-output repertoire. In what follows we explain how ϕ_δ, under the view of an algorithmic perturbation calculus [**?**] can help compute ϕ faster.

IIT is considered an essential part of a set of theories that contribute to the current discussion of consciousness but we are here only interested in phi as a quantitative measure for integrated information.

In this context, ϕ_δ is an alternative parameter. Whereas ϕ is calculated using statistical methods, ϕ_δ is based on premises of algorithmic complexity and simulation, this is, ϕ_δ exploits the idea that causal deterministic systems have a simple algorithmic description and thus a simple generating rule able to simulate and reproduce the complex behaviour of the system in question.

ϕ_δ is thus introduced as an alternative to other approaches to ϕ, with the chief difference that ϕ_δ exploits connections to algorithmic complexity.

While ϕ is calculated using statistical techniques, ϕ_δ is based on the direct connection between integrated information and algorithmic complexity.

In order to deduce such simple rules in systems of interest, we apply the perturbation test suggested in [10, **?**] in order to know the computational capabilities of them.

A perturbation test is applied first to deduce what the computational capabilities of a system are, and once this capabilities get clear, simple rules are formalized and implemented to simulate behaviour of these systems. Secondly, a kind of (automatic) meta perturbation test is applied over the behaviour obtained by last simple rules in order to obtain explanations of such behaviour.

2 Integrated Information and Algorithmic Complexity

Connections to algorithmic complexity has been done in the past, starting from Tononi's own first application of his own ϕ in a clinical context [**?**] where once ITT is introduced his group decided to use a lossless compression algorithm. Here we contribute to the formalization of such connection.

2.1 Causality at the intersection

XXX

3 Methodology

3.1 ϕ_δ simplicity v complexity test

ϕ_δ follows strictly IIT to compute integrated information. This is, as Oizumi and Tononi mentioned in [6], integrated information can be measured, roughly speaking, as distances between probability distributions that characterize a MIP (Minimum Information Partition), this is "the partition of [a system] that makes the least difference" [6].

ϕ uses what they have called "intrinsic information bottleneck principle" [7], this is, an exhaustive search for MIP among all possible partitions of a system, which makes ϕ computation to requires superexponential computational resources.

In contrast, ϕ_δ borns as an empirical experimentation of natural computation claims based on simple rules in the context of the study of mind. In consequence, we obtained a method that speed up computation of integrated information.

Discovering the simple rules that governs a "discrete dynamical system" [5] like those studied in IIT, in the context of natural computationalism, presuppose the analysis of its general behaviour pursuing a dual purpose: to know computational its capabilities and to obtain explanations and/or descriptions of the behaviour of the system itself. Such goals were reached by means the application of perturbation test.

In the next sections perturbation test and its application to implement ϕ_δ is explained in detail.

3.2 The programability test

In [10] it was proposed what he called a programmability test. Under the view that the universe and all physical systems living in it enabled to process information can be considered a (natural) computer [11] with particular computational capabilities [10]. Proposed was that in the same way as Turing test proceeds to ask questions to a computer in order to know if it is capable compute an intelligent behaviour. The test aims to know what is capable to compute an specific system by means asking questions.

Capabilities of a target system can be known analyzing the information content of the answers obtained during the test [12].

In practice, what the original system's perturbation test [**?**] suggested is to ask questions to a computational system in the form: *what is your output (answer) given this question (input)?*. This idea is revisited and adapted in the ϕ_δ's implementation, where the set of answers offered by the system should not just to offer a picture on its computability capabilities, but also on the behaviour of the system itself. In other hand, for a system be capable to respond questions,

it has to be enabled to do it. Such explanatory interface in ϕ_δ is based on simple embedding behaviour rules. All details on this are explained in following sections.

The discovery of simple rules used for the calculation of ϕ_δ are used exclusively to compose probability distributions used in IIT, the rest of the calculus remains the same as specified in IIT 3.0.

A key concept in IIT is the *constrained/unconstrained distributions* and *EMD distances* between them, from where the concepts *causal and effect information* derivates [4].

These concepts are shown in Figure 1 with the same three node example used in [6] by Oizumi and Tononi.

ϕ_δ uses the concept of Unconstrained Bit Probability Distribution (UBPD) in order to compute IIT's unconstrained/constrained probability distribution. In short, UBPD correspond to individual probabilities associated to a node of a system to take values of 1 (ON) or 0 (OFF). This concept is explained in Table 1, where UBPD are computed for the system show in Figure 1.

As can be noted in Table 1, after the definition of the system, outputs for each node are computed separately. As result we obtained individual probabilities for each node to take vales 0/1.

In contrast, ϕ_δ use a combination between perturbation test and simple rules for simulate behaviour of system and to obtain explanations of such behaviour.

As it was said before, it was necessary to apply a perturbation test in the previous analysis in order to find the simple rules that governing the system's behaviour. Then ϕ_δ uses an Zenil-test-like automatic process in order to (1) simulate behaviour, (2) find explanations of the behaviour and (3) use such answers to make calculations.

In the very first application of the perturbation test was found that information distribution along the behaviour of a system obeys a fractal distribution, what makes easier to express and compress behaviour. All details on this process are show in next sections.

4 Numerical Results

4.1 Causality, algorithmic complexity and integrated information

Examples, low v high K. etc

4.2 Finding simple rules in complex behaviour

A perturbation test was applied to systems which IIT is interested in, the set of answers were analyzed and generalized in order to find the rules that defines computability power of them, this is, rules that give an account of what the system can, and cannot compute. Finally, same test was applied to rules that

```
am = {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}};
dyn = {"OR", "AND", "XOR"};
calcUBPOutputs[1, am, dyn] //AbsoluteTiming
calcUBPOutputs[2, am, dyn] //AbsoluteTiming
calcUBPOutputs[3, am, dyn] //AbsoluteTiming
```

```
{0.000575, <|"ZeroProb" -> 0.25, "OneProb" -> 0.75|>}
{0.000287, <|"ZeroProb" -> 0.75, "OneProb" -> 0.25|>}
{0.000341, <|"ZeroProb" -> 0.5, "OneProb" -> 0.5|>}
```

Table 1: Computing UBPD for system shown in Figure 1

describe behaviour of the same system. For sake of clarity, the following recipe was applied to make possible ϕ_δ implementation.

1. perturbation test was applied to systems used in IIT to obtain detailed descriptions of behaviour of systems.

2. Results in step one were analyzed in order to reduce the dynamic of a system to a set of simple rules. This is, in the very sense of the natural computation claims, we found short cuts for simulation of systems behaviour, simple rules that can be used to know if a system is capable or is not capable to compute certain output without calculate the whole output repertoire.

3. A meta analysis was applied over behaviour of systems, this is, in the very same form of perturbation test, we looked for rules that describe the behaviour of interacting rules found in step 2.

4. Once rules in steps 2 and 3 were formalized, ϕ_δ was turned in a kind of interrogator whose purpose is to ask questions to a system related with its own computational capabilities and about its own behaviour.

For the sake of simplicity, ϕ_δ does not compute the whole output repertoire for a system due the very behaviour of the system is embedded in a set of simple rules. Then ϕ_δ is turned out in a kind of automatic interrogator that, in imitation of the perturbation test, ask questions in the for of: *tell me how information is distributed along all the patterns of your whole behaviour?* or for sake of simplicity; *tell me if you are capable to have this specific behaviour (pattern), and if you, tell me where, in the map of all your behaviour I can find it.*

To work in this way allows to ϕ_δ to speed up the integrated information for large systems, compared with the original version of IIT 3.0 [1].

This kind of analysis allowed us to find that information distribution in the complex behaviour of systems analyzed in IIT follows a fractal distribution

[1] There are research that keep looking for efficient and faster manners that have resulted successful, for instance [2]

susceptible to be summarized in simple rules, the finger print of fractals. This also make possible to find compressed forms to express answers given by the system when they are asked for explanations of its behaviour.

4.3 Simple rules and fractal distribution of information

As shown in [12], despite deriving from a very simple program, without knowing the source code of the program, a partial view and a limited number of observations can be misleading or uninformative illustrating how difficult it can be to reverse engineer a system from its evolution in order to discover its generating code [12].

In the context of IIT, when we talk about a complex network we find that there are different levels of understanding complex phenomena such as to know the rules performed by each node in a system, and to find the rules that describe its behaviour along the time. To achieve the second, as perhaps it could be done for the "whole scientific practice" [12], we found useful to perform programmability tests in order to reduce the behaviour of the subject systems. Results were analyzed and we found a fractal distribution of information in the behaviour of this kind of systems. Then as was to be expected, fractal behaviours are susceptible to be expressed with simple formula.

In order to explain how simple rues works for ϕ_δ, consider the system show in Figure 3. At the same time table 2 contains calculus of output repertoire for this last system.

Table 2 shows the whole behaviour (output repertoire) of the system which exhibit what can be referred to as a chaotic behaviour that can misled an observer where a pattern of regularities with subtle variations can be noticed. This perspective remains when when we take a look into isolated behaviours, for example, in table 3 where behaviour of nodes {4, 5} (columns 5 and 6 in results shown in table 2) are shown.

It can be seen that isolated behaviours still follow a sort of order. Such regular patterns were easily summarized in what we call behaviour tables, like those shown in Figure 4.

Last rows in behaviour tables shown in Figure 4 (using curly brackets) correspond to compressed representation of behaviours in shown in table 3, for nodes {4, 5} of the system in Figure 3.

Compressed expressions of behaviour for node 4, for instance, should be read as: 85 repetitions of 0 digit, followed by a pattern repeated 2 times, this pattern is: one time 1, followed of one digit 0 (this is {1->1, 1->0}). This pattern is followed for four digits 0 and so on. Here notice that the first number 85 is the sum of the numbers shown in 3th column of its behaviour table.

For node 5, compressed representation of behaviour should be read as: four times digit 0, followed by four digits 1. This pattern is followed by eight repetitions of digit 1. Such last patter is followed for 94 (32+64) repetitions of digit 1.

Representation used in this isolation of behaviours is expressed in terms of the nodes that "feed" to target nodes of this example (first left column in

Dynamic	Nodes	1	2	3	4	5	6	7
AND	1	0	0	1	0	0	0	1
OR	2	0	0	1	0	0	1	0
OR	3	1	0	0	0	1	0	1
AND	4	1	0	1	0	1	0	1
OR	5	0	0	1	1	0	1	1
OR	6	1	1	1	0	0	0	0
AND	7	0	1	0	1	1	1	0

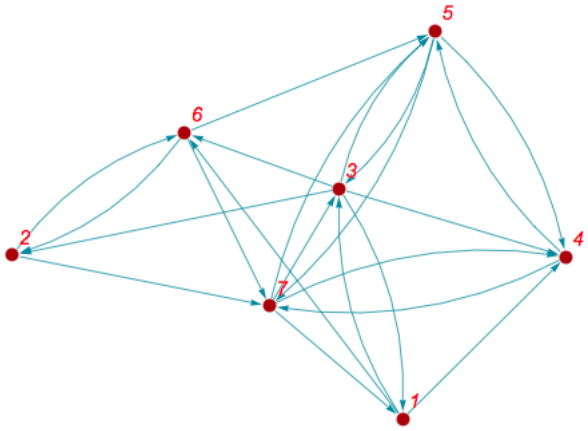

Figure 1: Seven nodes system

OVER CURRENT STATE (NODE 4. AND)			
Node	node-1=pow	2^(pow-1)	(n+1)/n
1	0	1	
3	2	4	4
5	4	16	4
7	6	64	4
85->0,{{{1->1,1->0}->2,4->0}->2},16->0			
{{{1->1,1->0}->2,4->0}->2			

OVER CURRENT STATE (NODE 5. OR)			
Node	node-1=pow	2^(pow-1)	(n+1)/n
3	2	4	
4	3	8	2
6	5	32	4
7	6	64	2
{{{4->0,4->1},8->1}->2},32->1,64->1}			

Figure 2: Tables behaviours for seven system in Figure 2. (A) node 4, (B) node 5. From left to right, Node column list input-nodes that feed the target node. node-1=power column computes the power used to transform from binary to decimal a pattern in the world of the 7 nodes systems. 2^pow column is decimal transformation operation. Forth column contains division between n+1 element in column 3 divided by element n

```
cm07 = {{0, 0, 1, 0, 0, 0, 1},
{0, 0, 1, 0, 0, 1, 0},
{1, 0, 0, 0, 1, 0, 1},
{1, 0, 1, 0, 1, 0, 1},
{0, 0, 1, 1, 0, 1, 1},
{1, 1, 1, 0, 0, 0, 0},
{0, 1, 0, 1, 1, 1, 0}};
dyn07 = {"AND", "OR", "OR", "AND", "OR", "OR", "AND"};
analysis07=runDynamic[cm07, dyn07]["RepertoireOutputs"]
```

```
{{0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 1, 0}, {0, 0, 0, 0, 0, 1, 0}, {0, 0, 1, 0, 0, 1, 0},
 {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 0, 0, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 0, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 0, 1, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 1, 0}, {0, 0, 1, 0, 0, 1, 0}, {0, 0, 1, 0, 0, 1, 0},
 {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 0, 1, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 0, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 0, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 1}, {0, 1, 1, 0, 1, 1, 1},
 {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 1}, {0, 1, 1, 0, 1, 1, 1},
 {0, 0, 1, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0},
 {0, 0, 1, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0},
 {0, 0, 1, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0},
 {0, 0, 1, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 0},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0},
 {0, 1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 1, 0, 1, 1, 1}, {0, 1, 1, 0, 1, 1, 1},
 {1, 1, 1, 0, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 0}, {1, 1, 1, 0, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1}}
```

Table 2: Seven nodes system and output repertoires

```
analysis07[[All, 4]]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1}
```

```
analysis07[[All, 5]]
```

```
{0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Table 3: List code 1

behaviour tables), namely nodes {3, 4} whose inputs according with Figure 3 are, for node 4: {1,3,5,7}, and for node 5: {3,4,6,7}.

This shallow analysis rises the intuition that behaviour of an isolated node can be expressed as "simple" rules in terms of its inputs. Obviously the whole behaviour of a system is not about isolated elements, but about elements interacting in a non linear manner. This last broader view is review in following sections. In this, we will focus, for sake of explanation, on behaviours of isolated nodes.

4.4 Limitations

When we visualize the behaviour of a system (or subsystem like an isolated node), and we take account its implementation, from the point of view of optimization of computational sources, to let run rules to generate the whole behaviour still being a challenge because it is an expensive process in terms of time and memory, then, for large systems, analysis based on exhaustive reviews of such behaviour it could eventually become intractable.

In order to overcome this limitation, ϕ_δ turned to find rules that not only give an account of computability capabilities of a system, but also to describe its own behaviour, or in other words, we wanted to know about possibilities to found "shortcuts to express the behaviour" [9] of a whole system.

In table 4 is shown how ϕ_δ implements this kind of shortcuts specifically for node 4 of the system shown in Figure 3, whose behaviour correspond to the

```
cm07 = {{0, 0, 1, 0, 0, 0, 1},
{0, 0, 1, 0, 0, 1, 0},
{1, 0, 0, 0, 1, 0, 1},
{1, 0, 1, 0, 1, 0, 1},
{0, 0, 1, 1, 0, 1, 1},
{1, 1, 1, 0, 0, 0, 0},
{0, 1, 0, 1, 1, 1, 0}};
dyn07 = {"AND", "OR", "OR", "AND", "OR", "OR", "AND"};
(*computing places in output repertoire where node 4 = 0*)
res070=onPossibleBehaviour[{4}, {0}, dyn07, cm07]
(*Sumarized representation of fractal behaviour*)
gp=givePlaces[res070["DecimalRepertoire"],res070["Sumandos"]];
```

```
<|"DecimalRepertoire" -> {0, 1, 4, 5, 16, 17, 20, 21, 64, 65,
    68, 69,80, 81, 84},
"Sumandos" -> {0, 2, 8, 10, 32, 34, 40, 42}|>
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 86, 88, 89,
 90, 91, 92, 94, 96, 97, 98, 99, 100, 101, 102, 103, 104,
 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
 116, 118, 120, 121, 122, 123, 124, 126}
```

Table 4: ϕ_δ asking for accounts of information distribution in behaviour of 4th node of the system show in Figure 3. (a) ϕ_δ's code asking for zero digit distribution in the whole behaviour, (b) compressed answer given by the system, (c) unfolded answer

first list of code show in table 3, corresponding at the same time to the first behaviour table in Figure 4.

In table 4 the expression that resume distribution of information, specifically distribution of zero digit in in the behaviour of node 4 of the system defined in Figure 3. To understand information distribution some specifications are needed.

First, recall above sections about compressed forms of behaviour, where was easy to see that digits that define behaviour of a node follow series of repetitions. A clear example is the sum of numbers in 3th column of behaviour table of node 4 in Figure 4, where such sum of powers (a function of input-nodes) is equal to number of repetitions of digit 0 at the beginning of the chain of digits that define behaviour of this node.

Implementation of such compressed expressions of behaviour allows us to

```
1 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1}
```

```
1 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
     18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
    32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
    46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
    60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
    74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 86, 88, 89,
    90, 91, 92, 94, 96, 97, 98, 99, 100, 101, 102, 103, 104,
    105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
    116, 118, 120, 121, 122, 123, 124, 126}
```

Table 5: Comparison between fractal rule of behaviour and real behaviour of node 4 of the system defined in Figure 3

express behaviour of a node in terms of inputs and to give explanations of such behaviour in terms of locations in the whole output repertoires, or in terms of ordinals.

Since ϕ_δ uses an automatic version of perturbation test to ask to a system about its own behaviour its answers are expressed in terms of ordinals. How automatic test works is explained in following sections, but at the moment, line 10 in list code shown in table 4 shows an example of its implementation, where ϕ_δ is ask: *for the node 4 in the system of seven nodes here defined, tell me how digit 0 is distributed.*

The system reacts to the ϕ_δ's question and it "answers" with a compressed form of this answer. Distribution of information in the answer follows a fractal nature, and as such, can be expressed with tiny rules. Such fractal expression is defined, as can be seen in section (c) in table 4, by two variables: *Decimal-Repertoire* that holds points where the patters defined by *Sumandos* variable must be self-repeated. This means that in order to unfold whole distribution (of zero digit), patter of numbers in *Sumandos* must be added to each value in *DecimalRepertoire*.

Once this fractal rule is infold we obtain the ordinal places where, in the whole behaviour of a node, digit 0 (or 1) can be found.

In table 5 the unfolded distribution of zero digit is compared with real behaviour of the node 4. Here, note that answer given by the system correspond to the ordinal places where zero digit is localized in the whole behaviour of node 4.

The task of finding information distribution and its implementation as simple

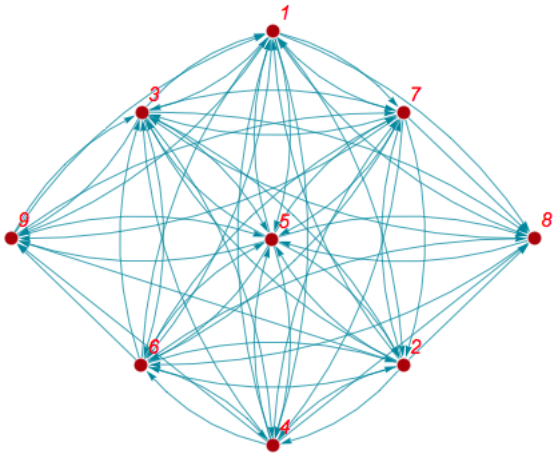

Figure 3: Network example with 9 nodes

rules as function of inputs of nodes is the result of complexity and algorithmic analysis, that basically was performed by means the application of perturbation test.

While this analysis were applied, we noticed that this test can be automatized in order to compute integrated information. Next section explains this.

4.4.1 Automatic perturbation test

The programability test suggested before [**?**, **?**] based on questions asked to know about the programmability capabilities was applied in a form of meta test where questions were "shot" to "force" the system to explain its own behaviour.

In last section was shown that the systems implemented as simple rules that rise complex behaviour enable to the system itself to "respond" where, in the chain of digits that conform its behaviour (of a specific node), certain pattern is. And due the fractal nature of information distribution in behaviour allows us to answer complex distribution in short shapes. In this section we will show advantages to use (automatic) perturbation test based over simulation of behaviour using simple rules over original version of IIT 3.0 based on "bottleneck principle" [7] to compute integrated information.

In the original perturbation test, questions take the form: *what is your output (answer) given this question (input)?* In ϕ_δ, since questions look for explanations of the behaviour of the system itself, they take the form: *tell me if you are capable to perform this specific behaviour (pattern), and if you, tell me where, in the map of all your behaviour is possible to find it.*

One example how this automatic implementation works is shown in table 6 where such test is applied for system shown in Figure 5.

In line 2, in the list code show in table 6, it is possible to see how ϕ_δ ask questions to a system. This line should be interpreted as, *Can compute the*

```
1 mmu=MemoryInUse[];
2 findPatternInSharedInputs[{1, 3, 5, 6}, "OR", 1, {1, 5, 7, 3},
      "AND", 1]//AbsoluteTiming
3 MemoryInUse[]-mmu
```

```
1 {{1, 1, 1, 0, 1}, {1, 1, 1, 1, 1}}}
2 0.000736
3 2440
```

Table 6: Asking questions automatically to system shown in Figure 5

pattern {8,9} = {1,1} when {8,9}-> {"OR", "AND"}? If yes, tell me in what conditions you can do that.

In this example, in the first place ϕ_δ tries to find needed conditions needed to compute an specific output. As table 6 shows, the answer is: *yes! I can. This may happen when {1,3,5,6,7} = {{1,1,1,0,1},{1,1,1,1,1}}*. In this answer {1,3,5,6,7} is the set of inputs to the subsystem {8,9}.

Notice here that answer offered by the system actually are the needed condition or necessary inputs to the system to compute an specific input, in the form of compressed schemas . The whole schemas are: {{1,*,1,*,1,0,1,*,*}, {1,*,1,*,1,1,1,*,*}}, where '*' is a wildcard that means 0/1 (any symbol), Note that such schemas correspond exactly to the generalized answer offered by the system, this is: {1,3,5,6,7} = {{1,1,1,0,1},{1,1,1,1,1}}.

This answer, in same way how the Holland's schema theorem [1] imitates genetics where a set of genes are responsible of specific features in phenotypes, what ϕ_δ retrieves is the general information that yields to specific inputs for the current system.

In order to explain advantages of the generalization of information in form of schemas computed by simple rules take a look to table 7, where the all possible cases were wanted pattern can be found in the whole output repertoire of the system here analyzed. In the second column in table 7 are all outputs where wanted output pattern {8,9}={1,1} is found (marked in red), in the left side are the 9-length inputs that yields to outputs that contains the wanted pattern. In inputs, in bold, are marked the corresponding inputs that are particularly responsible of cause the wanted pattern, this is, all possible patterns for the inputs {1,3,5,6,7}

In order to obtain results in table 7 was necessary to define the whole set of all 2^9 possible inputs and compute the whole set of outputs, then an exhaustive search for the wanted pattern {8,9}={1,1} was realized. Notice that time and memory used are, at least, 10 times bigger than the used by ϕ_δ approach, these are the last two rows show in table 6 and table 7.

Compression and generalization of systems, its behaviour and information distribution based on simple rules, perhaps the stronger claim of natural computation results in advantages over common sense or classical approaches for

analysis of complex systems, particularly in terms of computational sources needed to compute integrated information.

In summary, in this section was shown that, ϕ_δ implementation is based on representation of systems using simple rules which take the advantage that information is propagated as patters that are self replicated in specific points along the map of the behaviour of a system, this is a fractal distribution of information to offer complex explanations of the behaviour itself in a compressed form, this is, in terms of ordinals. This kind of computations are possible after to know if a system is capable to compute a pattern and under what conditions.

All above was applied for isolated and simple cases of two bytes patterns. In the next section generalization for n nodes of the system is addressed and how this works to compute integrated information according to IIT.

4.5 Shrinking after dividing to rule

In the last sections it has been shown how ϕ_δ, applying perturbation test can deduce firstly what is capable a system to compute and if it is, under what conditions a computation can be performed, and secondly, that by means simple rules specification of system is possible to obtain a descriptions of the behaviour of it in the form of rules that say how information is distributed, or in other words, where, in ordinal terms, such conditions can be found.

The ultimate objective of obtaining this kind of descriptions of the behaviour of the system, is to know how many times specific patterns appears in whole repertoires to construct probability distributions without the need for consumption of exaggerated computational resources, since these probability distributions are a key piece used by IIT to compute integrated information.

ϕ_δ advanced on such challenges using a two-pronged strategy consisting firstly on parallelizing analysis process, which is not more than a technical strategy available to be implemented in almost any computer language and that falls beyond the scope of this paper, and secondly by means partition of target sets. This last part of ϕ_δ's strategy consist in, given a target set to be analyzed, this is partitioned into parts to be interrogated each by ϕ_δ using implementation of automatic test.

In this approach it is easy to see how, when a partition of the current target set has been analyzed, the search space for the remaining parts is significantly reduced facilitating and accelerating the analysis of the remaining partitions.

In order to illustrate last idea, take for example the table 8.

Table 8 shows definition of a system of 7 nodes, where a set of a progressively growing length is analyzed. In this example ϕ_δ ask repeatedly to the system if is capable to find a growing pattern of zeros, and if it is capable, the system is requested for showing where is possible to find the wanted pattern. Obviously larger patters need a bigger amount of computations, but as can be seen in table 8, the first data of results shown that time used by ϕ_δ as pattern's length gets larger, it grows linearly in contrast with IIT 3.0 that grows exponentially.

{**1**,0,**1**,0,**1**,**0**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**0**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**0**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**0**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**1**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**1**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**1**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**1**,**1**,0,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**0**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**0**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**0**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**0**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**1**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**1**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**1**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**1**,**1**,1,0}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**0**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**0**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**0**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**0**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**1**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**1**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**1**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**1**,**1**,0,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**0**,**1**,1,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**0**,**1**,1,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**0**,**1**,1,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**0**,**1**,1,1}->{1,1,0,1,0,1,1,**1**,**1**},
{**1**,0,**1**,0,**1**,**1**,**1**,1,1}->{1,1,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,0,**1**,**1**,**1**,1,1}->{1,1,0,0,0,0,1,**1**,**1**},
{**1**,0,**1**,1,**1**,**1**,**1**,1,1}->{1,0,0,1,0,0,1,**1**,**1**},
{**1**,1,**1**,1,**1**,**1**,**1**,1,1}->{1,0,1,0,1,1,1,**1**,**1**},
0.166334
217920

Table 7: Naive approach of looking for a pattern

```
cm07 = {
{0, 0, 1, 0, 0, 0, 1},
{0, 0, 1, 0, 0, 1, 0},
{1, 0, 0, 0, 1, 0, 1},
{1, 0, 1, 0, 1, 0, 1},
{0, 0, 1, 1, 0, 1, 1},
{1, 1, 1, 0, 0, 0, 0},
{0, 1, 0, 1, 1, 1, 0}};
dyn07 = {"AND", "OR", "OR", "AND", "OR", "OR", "AND"};
onPossibleBehaviour[{1,2,3},{0,0,0},dyn07,cm07]//
    AbsoluteTiming
onPossibleBehaviour[{2,3,4},{0,0,0},dyn07,cm07]//
    AbsoluteTiming
onPossibleBehaviour[{1,2,3,4},{0,0,0,0},dyn07,cm07]//
    AbsoluteTiming
onPossibleBehaviour[{5,6,7},{0,0,0},dyn07,cm07]//
    AbsoluteTiming
onPossibleBehaviour[{1,2,3,4,5,6,7},{0,0,0,0,0,0,0},dyn07,cm07
    ]//AbsoluteTiming
```

```
{0.00076,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
{0.000647,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
{0.0009,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
{0.000656,<|"DecimalRepertoire"->{0,16},"Sumandos"->{0}|>}
{0.001248,<|"DecimalRepertoire"->{0},"Sumandos"->{0}|>}
```

Table 8: Comparing.

5 Conclusion and future directions

This paper focused on the calculus of probability distribution defined by IIT 3.0. It remains to make similar and detailed connections to what Oizumi and Tononi et al. called MIP (Minimum Information Partition) [7] of a system. But with this first version of ϕ_δ we suspect that MIP definitions obey also the same rules of algorithmic nature so the next step is to go further in application of the test introduced in this paper to find simple rules that helps to find MIP in a more natural and faster way.

Natural computation allows implementation based on simple rules that appeals to emulation/simulation of natural behaviour of systems unlike more classical approaches based on the calculation of entire repertoires of inputs and outputs. This results on savings of computational resources needed to perform analysis or large network systems.

We conclude that distribution information in networks here analyzed follows a fractal distribution susceptible to be expressed in compressed forms due its algorithmic nature.

The perturbation programmability test, initially inspired by the Turing test applied to physical systems is a working strategy to find explanations about the behaviour of systems.

References

[1] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[2] Jun Kitazono, Ryota Kanai, and Masafumi Oizumi. Efficient Algorithms for Searching the Minimum Information Partition in Integrated Information Theory. *Entropy*, 20(3):173, March 2018. arXiv: 1712.06745.

[3] Stephan Krohn and Dirk Ostwald. Computing integrated information. *Neuroscience of Consciousness*, 2017(1), January 2017.

[4] William Marshall, Jaime Gomez-Ramirez, and Giulio Tononi. Integrated Information and State Differentiation. *Frontiers in Psychology*, 7, June 2016.

[5] William G. P. Mayner, William Marshall, Larissa Albantakis, Graham Findlay, Robert Marchman, and Giulio Tononi. PyPhi: A toolbox for integrated information theory. *arXiv:1712.09644 [cs, q-bio]*, December 2017. arXiv: 1712.09644.

[6] Masafumi Oizumi, Larissa Albantakis, and Giulio Tononi. From the phenomenology to the mechanisms of consciousness: integrated information theory 3.0. *PLoS Comput Biol*, 10(5):e1003588, 2014.

[7] Masafumi Oizumi, Larissa Albantakis, and Giulio Tononi. From the phenomenology to the mechanisms of consciousness: integrated information theory 3.0. *PLoS Comput Biol*, 10(5):e1003588, 2014.

[8] Giulio Tononi. An information integration theory of consciousness. *BMC neuroscience*, 5(1):42, 2004.

[9] Stephen Wolfram. What Is Ultimately Possible in Physics? pages 417–433, 2012.

[10] Hector Zenil. A behavioural foundation for natural computing and a programmability test. In *Computing Nature*, pages 87–113. Springer, 2013.

[11] Hector Zenil. *A computable universe: understanding and exploring nature as computation*. World Scientific, 2013.

[12] Hector Zenil, Angelika Schmidt, and Jesper Tegnér. Causality, information and biological computation: An algorithmic software approach to life, disease and the immune system. *arXiv preprint arXiv:1508.06538*, 2015.