Abstract

Integrated information has been introduced as a metric to quantify the amount of information generated by a system beyond the information generated by its individual elements. While the metrics associated with the Greek letter ϕ require the calculation of the interaction of an exponential number of sub-divisions of the system, most of these numerical approaches related to the metric are based on the basics of classical information theory and perturbation analysis. Here we introduce and sketch alternative approaches to connect algorithmic complexity and integrated information based on the concept of algorithmic perturbation rooted in algorithmic information dynamics and its concept of programmability. We hypothesise that if an object is algorithmic random or algorithmic simple, algorithmic random perturbations will have little to no effect to the internal capabilities of a system to produce integrated information but when an object is more integrated the object will also display elements able to perturb the object and increase or decrease its algorithmic randomness. We sketch some of these ideas related to an object integrated information value and its algorithmic information content. We propose that such an algorithmic perturbation test quantifying compression sensitivity may provide a system with a means to extract explanations–causal accounts–of its own behaviour hence making IIT and associated measure ϕ more explainable and interpretable. Our technique may reduce the number of calculations to arrive at some estimations with algorithmic perturbation guiding a more efficient search. Our work sets the stage for a systematic exploration and further investigation of the connections between algorithmic complexity and integrated information at the level of both theory and practice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of information has emerged as a language in its own right, bridging several disciplines that analyze natural phenomena and man-made systems.

The development of techniques to decipher the structure and dynamics of complex systems is a rich inter-disciplinary research area which is not only of fundamental interest but also important in numerous applications. Broadly speaking, dynamical aspects such as stability and state-transitions within such systems have been of major interest in statistical physics, dynamical systems, and computational neuroscience (6, 7, 27). Here, complex systems are defined by a set of non-linear evolution equations. Cellular automata, spin-glass systems, Hopfield networks, and Boolean networks, have for example been used as numerical experimental model systems to investigate the dynamical aspects of complex systems. Due to the complexity of the analysis, notions such as symmetries in the systems, averaging (e.g. mean-field techniques), and separation of time-scales, have all been instrumental in deciphering the core principles at work in such complex systems. In parallel, network science has emerged as a rich interdisciplinary field, essentially analyzing the structure of real networks in different areas of science and in diverse application domains (3). Examples include social, biological and electrical networks, the web, business networks and the interconnected internet. By a structural analysis, which has dominated these investigations, we refer to statistical descriptions of network connectivity. Networks can be described globally, in terms ranging from the degree to which they differ from a random Poisson distribution of links, to their modular organization, including their local properties such as local clustering around nodes, special nodes/links with high degrees of betweenness or serving specific roles in the network, and local motif structures. Such network features can be used to classify and describe similarities and differences between what appear to be different classes of networks across and within different application domains. Finally, due to the rich representational capacity of networks and their usefulness across science, technology, and applications, work in machine learning, in particular graph convolutional networks and embedding techniques, is currently making headway in devising ways to map these non-regular network objects onto a format such that machine learning techniques can be used to analyze their properties (4).

Now, we may ask if integrated information theory (IIT) is proposed to be of relevance for the analysis of complex networks, we ask how is IIT related to fundamental questions underpinning research and thinking of complex systems? On the one hand, we find a rich body of work dealing with what could be referred to as technical, computational challenges, and application-driven investigations. For example, which global and local properties should be computed and how to do so in an efficient manner. However, at a more fundamental level we find essentially two challenges, which in our view have a bearing on the core intellectual driving force of complex systems. First: What is the origin of and mechanisms propelling order in complex systems? Secondly, and of major concern for the present paper: Is the whole - in some sense - larger than the sum of its parts? Both questions are vague when formulated in words, as above, but they can readily be technically specified within a model class. The motivation for the second question is that it appears that there are indeed phenomena in nature which cannot easily be explained only with reference to their parts, but seem to require that we adopt a holistic view. Since Anderson's classic 1972 essay, there has been an animated and at times heated discussion of whether there is anything which could be referred to as emergence.(1)

Tononi and his group have developed a formalism–targeting exactly such a holistic analysis–specifically to quantify the amount of information generated by a system – defined as a set of interconnected elements–beyond the information generated by the parts (subsets) of the system. Their motivation was that in order to develop a theory of consciousness (17). In that quest, they perceived a necessity to define a measure which could quantify the amount and degree of consciousness, a measure they refer to as ϕ, which in turn constitutes the core of Integrated Information Theory or IIT. Importantly, in the present work we distinguish between the issue of the relevance of ϕ for consciousness versus the technical numerical question of how to calculate ϕ. Here we address the computation of ϕ, as it is potentially a means toward a precise formulation for the possible causal relation between a whole and the parts of a system, regardless of its purported relevance to consciousness. To calculate ϕ, Tononi and collaborators have developed a computational toolbox (16). Yet, calculating ϕ comes with a severe computational cost, as the calculation scales exponentially with the number of elements in the network. Furthermore, the computation requires knowledge of the transition probabilities of the system, which makes computation of anything larger than small systems of order of one magnitude intractable in practice. The calculation of ϕ requires a division of the system into smaller subsets, ranging from large pieces down to singletons, every division into k pieces can be instantiated in $\binom{N}{k}$ different ways. Using this procedure from Tononi, elements that have small causal influences on the activity of other elements can be identified. A system with low ϕ is therefore characterized by the fact that changes in subsets of the system do not affect the rest of the system. Such a system is therefore considered to be a non-integrated system. This observation entails a key insight, namely, that if a system is highly integrated among its parts, then the different parts can be related to each other, or more precisely, they can be used to describe other parts of the system. Then the parts are in some sense simple and should

be compressible.

This is the observation and intuition behind our method, which employs a formalized notion of complexity to exploit this insight and thereby allow a more efficient, guided search in the space of algorithmic distances, in contrast to exhaustive computations of the distance between statistical distributions, as currently implemented in IIT. Technically we are therefore not required to perform a full computation of what is referred to as the input-output repertoire (see Methods for technical details). This, in brief, is our motivation for introducing our method, which is based on algorithmic information dynamics (15, 44, 45). At its core is a causal perturbation analysis and a measure of sophistication connected to algorithmic complexity. Our approach exploits the idea that causal deterministic systems have a simple algorithmic description and thus a simple generating mechanism sufficient to simulate and reproduce complex systemic behaviour. Using this technique we can assess the effect of perturbations, and thereby exploit the fact that, depending on the algorithmic complexity of a system, the perturbation will induce different degrees of change in algorithmic space. In short, a system will be highly integrated only if the removal or perturbation of its parts has a non-linear effect on the generative program producing the system in the first place.

Interestingly, even Tononi suggested early on that algorithmic complexity could be connected to the computation of integrated information (20). However, a lossless compression algorithm was used to approximate ϕ. Here we contribute to the formalization of such a suggestion by using stronger tools, which we have recently developed, to approximate complexity. At the core of algorithmic information is the concept of minimal program-size and Kolmogorov-Chaitin complexity (5, 14). Briefly, the Kolmogorov-Chaitin complexity $K(x)$ of an object x is the length of the shortest computer program that produces x and halts. $K(x)$ is uncomputable but can be approximated from above, meaning one can find upper bounds by using compression algorithms, or rather more powerful techniques such as those based on algorithmic probability (8, 25, 40), given that popular lossless compression algorithms are limited and more closely related to classical Shannon entropy than to K itself (38, 39, 42). One reason for this state of affairs is that, as demonstrated in (24), there is a fundamental difference between algorithmic and statistical complexity with respect to how randomness is characterised in opposition to causation. Specifically, algorithmic complexity implies a deterministic description of an object (it defines the algorithmic information content of an individual sequence/object), whereas statistical complexity implies a statistical description (it refers to an ensemble of sequences generated by a certain source). Approaches such as transfer entropy (22), Granger causality (10), and Partial Information Decomposition (33, 34) that are based on regression, correlation and/or a combination of regression, correlation and intervention but ultimately relying on probability distributions, fall into this category. Hence for better-founded methods and algorithms for estimating algorithmic complexity, we recommend the use of our tools, which are already being used by independent groups working on, for example, biological modelling (12), cognition (31) and consciousness (21). These tools are based on the theory of algorithmic probability, and are not free from challenges and limitations, but they are better connected to the

algorithmic side of algorithmic complexity, rather than only to the statistical pattern-matching side that current approaches using popular lossless compression algorithms exploit, making these approaches potentially misleading (42).

Our procedure, in brief, is as follows. First, we deduce the rules in systems of interest: we apply the perturbation test introduced in (15, 35, 44) to ascertain the computational capabilities of networks. Next, simple rules are formalized and implemented to simulate the behaviour of these systems. Following this analysis, we perform an automatic procedure, referred to as a meta-perturbation test, which is applied over the behaviour obtained by the aforementioned simple rules, in order to arrive at explanations of such behaviour. We incorporate the ideas of an interventionist calculus (c.f. Judea Pearl (19)) and perturbation analysis within what we call Algorithmic Information Dynamics, and we go beyond pattern identification using probability theory, classical statistics, and correlation analysis by developing a model-driven approach that is fed by data. This contrasts with a purely data-driven approach, and is a consequence of the fact that our analysis considers the algorithmic distance between models.

Chapter 2

Theoretical Framework

Integrated information theory (IIT) postulates that consciousness is identical to integrated information and that a system's capacity for consciousness can be expressed by a quantitative measure denoted by ϕ. Tononi defines integrated information as "the amount of information generated by a complex of elements, above and beyond the information generated by its parts" (29) and states, *"The integrated information theory* (IIT) of consciousness claims that, at a fundamental level, consciousness is integrated information" (29) (italics in original). IIT aims to explain "relationships between consciousness and the Physical Substrate of Consciousness (PSC), and starts from essential properties of phenomenal experience, and derives the requirements for the physical substrate of consciousness." (30)

Calculus of ϕ

The integrated information theory defines integrated information (ϕ) as the effective information of the minimum information partition (MIP) in a system (17, 18, 28, 30). The MIP is also defined as the partition having minimum effective information among all possible partitions.

$$\phi[X;x] =: \varphi[X;x,MIP(x)]$$

$$MIP(x) =: argmin\varphi(X;x,P)$$

Where X is the system, x is a state, and P is a partition $P = M_1, \ldots, M_r$.

Importantly, identifying the MIP requires searching all possible partitions and comparing their effective information ϕ. This effective information is specified in terms of effect and causal information, that is, the distance between two probability distributions: one for the unpartitioned (unconstrained) partition (this can be the full set of nodes of the whole system or one of its possible partitions) and a partition of this latter. Such probability distributions determine probabilities of all possible future (effect) or past (causal) states of an arbitrary partition being in a current state. This means that comparing one set of nodes that can be the full set of nodes of the system or a subset

(partition) of itself with all possible partitions of this set of nodes, MIP represents the partitions with the minimal value of the distance between probability distributions of the set of nodes and one of all its possible partitions.

When a set of nodes is chosen to compute effective information, this is referred to as a 'mechanism', and the partition to which it is compared is referred to as the 'purview'. The distance between probability distributions is computed by means of an adaptation of the Earth Mover's Distance (EMD) algorithm, which is a method to evaluate dissimilarity between two multi-dimensional distributions in a given feature space where a distance measure between single features, which we call the ground distance, is given. The EMD "lifts" this distance from individual features to full distributions. Note that EMD is referred to as a Wasserstein metric in mathematics, and is commonly used in machine learning as a natural metric between two distributions (32).

Intuitively, given two distributions, one can be seen as a mass of earth properly spread in space, the other as a collection of holes in that same space. Then, the EMD measures the least amount of work needed to fill the holes with earth. Here, a unit of work corresponds to transporting (by an optimal transport method) a unit of earth a unit of ground distance.

Chapter 3

Methods

In this section we introduction of the meta-perturbation analysis, additional technical details of which are presented in the Appendix. Next, we recap the causal perturbation and causal analysis leading up to the notion of program-size divergence, which is our core metric for how different programs, i.e. systems–more or less integrated–respond to perturbations

3.0.1 Programmability test and meta-test

In (35) a programmability test is introduced which was inspired by the Turing test, while being based on the view that the universe and all physical systems living in it and able to process information can be considered (natural) computers (35) equipped with particular computational capabilities (36).

The programmability test is explained as: "...replacing the question of whether a system is capable of digital computation with the question of whether a system can behave like a digital computer and whether a digital computer can exhibit the behaviour of a natural system."

Then, in the same way that the Turing test proceeds to ask questions of a computer in order to determine whether it is capable of computing an intelligent behaviour, the programmability test aims to know what a specific system is capable of computing by means of algorithmic querying (46).

In practice, the programmability test is a system perturbation test (35, 37) that "asks" questions of a computational system in the form: *what is your output (answer) given this question (input)?*. This idea is applied to ϕ_K's implementation so that once the set of all possible answers of a system is obtained, this set is analyzed and generalized to deduce the rules that should not just offer a picture of its computability capabilities, but also simulate and give an account of the behaviour of the system itself.

A second step after this perturbation test is to analyze its results in order to construct a computer program–as simple as possible–capable not only of reproducing the output repertoire but also of giving an account of the programmability capabilities of the system itself, that is, rules capable of producing a certain output given an input,

and at the same time explaining where, in ordinal terms, such an output could be placed relative to the order of the full output repertoire. This latter aspect we refer to as the meta-perturbation test.

Then, ϕ_K not only applies a perturbation test over a system, but also a meta-perturbation test over results obtained on the first test. The rules found in this meta-test are used not only as compressed specifications or representations of the behaviour itself, but also as rules that give a sort of account of the behaviour of the system.

This can be done because the systems analyzed in IIT are well known, or in other words, since all node-by-node operations are well defined, it is easy to compute all possible outputs (answers) for all possible inputs (questions or queries), corresponding to what in IIT are referred to as repertoires. In the context of ϕ_K, a meta-test is applied in order to find the rules that describe the behaviour embedded in repertoires of a system, instead of trying to ascertain the rules that define how the system works.

A system specified in this manner turns on a “computer", recording it's own behaviour (e.g the repertoires) as well as probing itself, e.g. the action of ϕ_K, in such a manner as to potentially give an account of its own behaviour. To make this possible a system specification must be enabled with an explanatory interface based on these simple embedding behaviour rules. ϕ_K goes beyond the original ϕ in that the programmability test searches for the rules underlying the behaviour of a system rather than generating a description of its possible causal connections. While in IIT these rules are defined a priori and induced by perturbation, ϕ_K's objective is not only to find rules that simulate, but also describe such behaviour in a brief manner (thus simple rules) and make predictions about the behaviour of the system. The field of Algorithmic Information Dynamics (15, 44) implements this approach by asking what changes to hypothesized outputs mean for the hypothesized underlying program generating an observation, after an induced or natural perturbation.

The simple rules discovered and used for the calculation of ϕ_K are used here exclusively to compose *constrained/unconstrained distributions* used in IIT for obtaining *cause-and-effect information*, a key concept from which the integration of information derives. The rest of the calculus–earth mover's distance measurements, the calculus of conceptual spaces, major complex and finding the MIP– remains as specified in IIT 3.0.

3.0.2 Causal perturbation analysis

From a statistical standpoint, it would be typical to suggest that the behaviour of two time series, let's call them X and Z, would potentially be causally connected if they were statistically correlated. Yet, there are several other cases that would not be distinguishable after a correlation test. A first possibility is that the time series simply shows similar behaviour without being causally connected, i.e. there is a shared upstream causal driver Y, concealed from the observer. Another possibility is that they are causally connected, but that correlation does not tell us whether it is a case of X affecting Z, or vice versa.

Perturbation analysis allows some disambiguation. The idea is to apply a perturba-

tion on one time series and see how the perturbation spreads to the other time series. Perturbing the data-point in position 5 the time series Z as shown in Figure 3.1 multiplying it by -2, X does not respond to the perturbation. This means that for this data point, X remains the same. This suggests that there is no causal influence of Z on X.

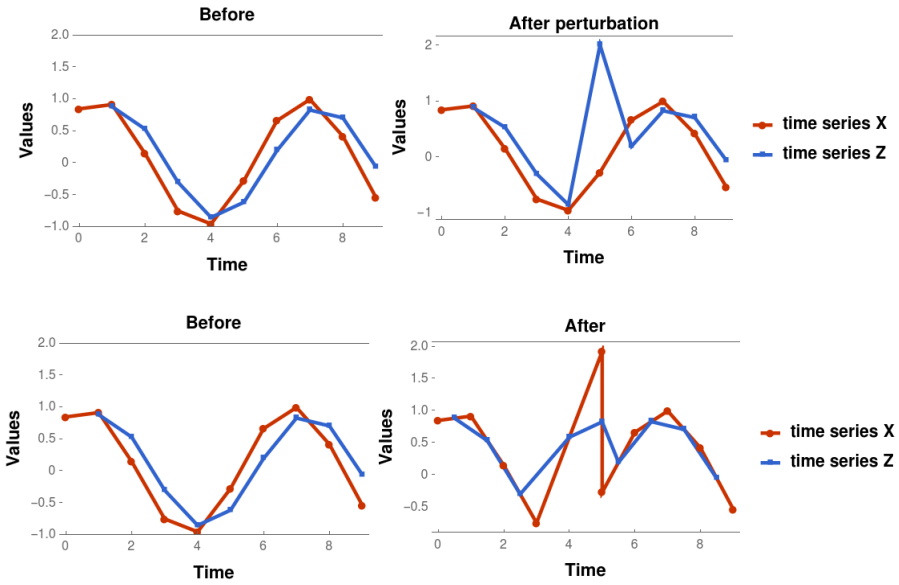

Figure 3.1: Causal intervention analysis on time series X and Z before and after perturbation in Z (top) and X (bottom). The values of Z come from the moving average of X, so there is a one-way causal relationship: perturbing X has an effect on Z but perturbing Z has no effect on X thereby suggesting the causal relationship.

In contrast, if the perturbation is applied to a value of X, Z changes and follows the direction of the new value, suggesting that the perturbation of X has a causal influence on Z. From behind the scenes, we can reveal that Z is the moving average of X, which means that each value of Z takes two values of X to calculate, and so is a function of X. The results of these perturbations produce evidence in favour of a causal relationship between these processes, if we did not know that they were related by the function we just described.

This suggests that it is X which causally precedes Z. So we can say that this single perturbation suggests a causal relationship illustrated in Figure 3.2.

Figure 3.2: Possible self-loopless causal relationship between two unlabelled variables X and Z.

There are a number of possible types of causal relationship between three events

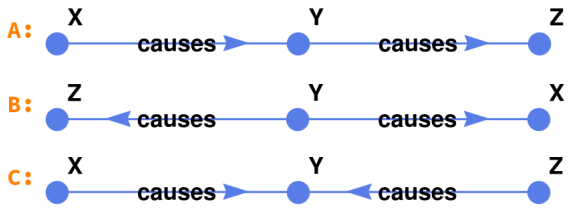

Figure 3.3: Acyclic path graphs representing all possible self-loopless connected causal relationships among 3 unlabelled variables.

(see Figure 3.3) that can be represented in what is known as a directed acyclic graph (DAG), that is, a graph that has arrows implying a cause and effect relationship but has no loops, because a loop would make a cause into the cause of itself, or an effect that is also its own cause, something that would be incommensurate with causality. In these graphs, nodes are events and events are linked to each other if there is a direct cause-and-effect relation.

In the first case, labelled A in orange, the event X is the cause of event Y, and Y is the cause of event Z, but X is said to be an indirect cause of Z. In general, we are, of course, always more interested in direct causes, because almost anything can be an indirect cause of anything else. In the second case B, an event Y is a direct cause of both Z and X. Finally, in case C, the event Y has 2 causes, X and Z. With an interventionist calculus such as the one performed on the time series above, one may rule out some but not all cases, but more importantly, the perturbation analysis offers the means to start constructing a model explaining the system and data rather than merely describing it in terms of simpler correlations.

In our approach to integrated information, the idea is to identify the set of most likely generating candidates able to produce certain observed behaviour even if such behaviour may not carry any statistical regularity and for all purposes appear statistically random (9). Strictly speaking, computational mechanics (23, 24), is a framework that bridges statistical inference and stochastic modelling that suggests a model based on an automaton called an ϵ-machine. However, such machines are stochastic in nature and, if the methods used to reconstruct such machines rely on statistical methods, the result is only an apparent causal representation with no correspondence between internal states and alleged states of the phenomenon observed. In contrast, approaches based on algorithmic probability as approached by algorithmic information dynamics can complement computational mechanics as they provide means to construct non-stochastic automata models that are independent of probability distributions and are in a strict sense optimal and universal (26).

In the case of our two time series experiments, the time series X is produced by the mathematical function $f(x) = Sin(x)$, and thus $Sin(x)$ is the generating mechanism of time series X. On the other hand, the generating mechanism of Z is $MovAvg(f(x))$, and clearly $MovAvg(f(x))$ depends on $f(x)$, which is $Sin(x)$, but $Sin(X)$ does not depend on $MovAvg(f(x))$. In the context of networks, the algorithmic-information

dynamics of a network is the trajectory of a network moving in algorithmic-information space together with the identification of those elements that shoot the network towards or away from randomness.

3.0.3 Causal influence and sublog program-size divergence

According to Algorithmic Information Dynamics (15, 44) there is an algorithmic causal relationship between two states s_t and $s_{t'}$ of a system M and M' if

$$|K(M_{s_t}) - K(M'_{s_{t'}})| \leq \log_2(t) + c$$

That is, if the descriptions of such systems can be bounded by $\log_2$ and a small constant c, then M is most likely equal to M' but in some other time state. In other words, if there is a causal influence of s_t on s_{t+1} or s_{t+1} on s_t as a system in isolation, their M and M' short descriptions should not differ by more than the description of the difference. However, if the descriptions of the states of a system (which may be two systems) in different alleged state times are not causally connected, their difference will diverge beyond above bound. In integrated information, causal influence among its parts is what is claimed to be measured and how different elements of a system can be explained by a single model or the other parts of the system informs us as to how integrated a system may be. A system characterized by large divergence is less integrated compared to a system which evolves with small differences in its respective subpart descriptions.

We will suggest that perturbations have to be algorithmic in nature because they need to be made or quantified at the level of the generating mechanisms from the whole or different parts of the integrated system and not at the level of the observations. For example, some $n-$ary expansions of the mathematical constant π according to BPP (named after Bailey-Borwein-Plouffe) formulas (2) allow perturbations to the digits that do not have any further effect because no previous digits are needed to calculate any other segment of π in the same base. The constant π then can be said to be information disintegrated to the extent of the BPP representations. Algorithmically low complexity objects have low integrated information. Similarly, highly random systems have low integrated information, because perturbations have little to no impact. Integrated information is, therefore, a measure of sophistication that separates high integration from both random and trivially non-random states.

3.0.4 A simplicity versus complexity test

With the previous section in mind we can proceed to introduce the idea of ϕ_K, where K stands for the letter often used for algorithmic (from Kolmogorov or Kolmogorov-Chaitin) complexity, and ϕ for the traditional of integrated information theory (17). The measure ϕ_K mostly follows methods that Oizumi and Tononi set forth in (17), where integrated information is measured, roughly speaking, as distances between probability

distributions that characterize a MIP (Minimum Information Partition), that is, "the partition of [a system] that makes the least difference" (17).

However, the difference between IIT's ϕ and ϕ_K lies in how ϕ_K circumvents what is called the "intrinsic information bottleneck principle" (18) that traditionally requires an exhaustive search for the MIP among all possible partitions of a system, a procedure responsible for the fact that integrated information computation requires super-exponential computational resources. In contrast to ϕ, which follows a statistical approach to estimating and exhaustively reviewing repertoires, the approach to ϕ_K is based on principles of algorithmic information.

Discovering the simple rules that govern a "discrete dynamical system" (16) like those studied in IIT presupposes the analysis of its general behaviour in pursuit of a dual agenda: first, to determine its computational capabilities, and secondly to obtain explanations and descriptions of the behaviour of the system.

As a consequence, one of the major adaptations of IIT is that ϕ_K uses the concept of Unconstrained Bit Probability Distribution (UBPD), that is, the individual probabilities associated with a node of a system taking values of 1 (ON) or 0 (OFF) after it has been "fed" all its possible inputs or after all possible perturbations.

In the context of ϕ_K, UBDP is estimated by approximating the algorithmic complexity of the TPM (Transition Probability Matrix) to compute IIT's unconstrained/-constrained probability distributions.

In Figure 3.4 and Table 3.1 the concept UBPD and its calculus is explained, using the example used by Oizumi et. al. in (17).

In order to explain the notion of UBPD we use FigureS 3.4, 3.5 and Table 3.1. in Figure 3.4 we use Oizumi's example used in (17) to calculate information integration. Figure 3.4-A shows the network representation: three nodes fully connected with different types of operation executed on its inputs, that is, for example, inputs to node A (coming from B and C nodes) will be processed in a logical OR operation. In Figure 3.4-B the adjacency matrix that represents the same same network is shown. This adjacency matrix uses the number 1 to indicate if a node receives signals (inputs) for another node. For example, the first row in the adjacency matrix indicates that node 1 or A receives inputs from nodes B and C, denoted as nodes 2 and 3. Finally, Figure 3.4-C shows the full input and output repertoires, that is, for the full set of all possible inputs to this system, all corresponding outputs are calculated according to the logical operations defined.

Table 3.1 shows code for computing UBPD for the system in Figure 3.4. This computation starts with the specification of the adjacency matrix (line 1) and internal dynamic (line 2) of the target system. Then, lines 1 and 2 in Table 3.1 represent code to network specified in Figures 3.4-A and 3.4-B.

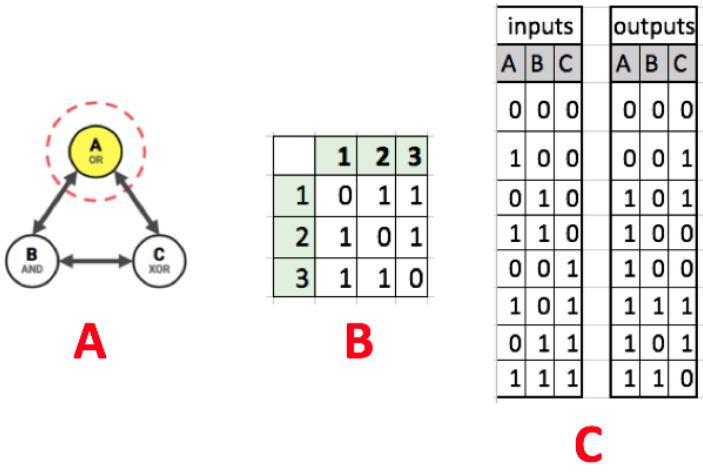

Figure 3.4: Three node example of a full connected network. From (17).

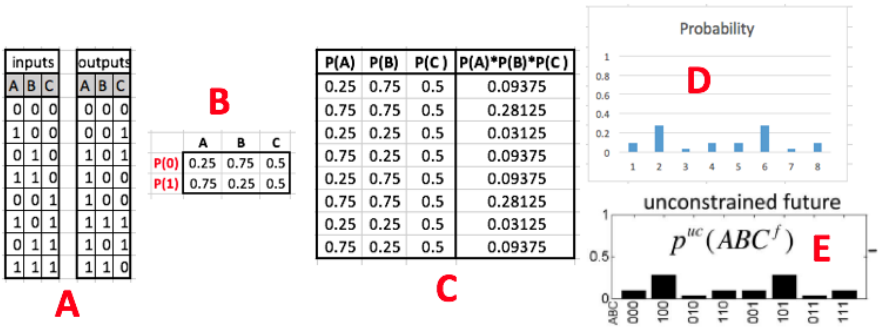

Figure 3.5: An example of using UBPC to calculate an unconstrained output distribution.

```
1  In := am = {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}};
2  In := dyn = {"OR", "AND", "XOR"};
3  In := calcUBPOutputs[1, am, dyn] // AbsoluteTiming
4  In := calcUBPOutputs[2, am, dyn] // AbsoluteTiming
5  In := calcUBPOutputs[3, am, dyn] // AbsoluteTiming
6
7  Out = {0.000575, <|"ZeroProb" -> 0.25, "OneProb" -> 0.75|>}
8  Out = {0.000287, <|"ZeroProb" -> 0.75, "OneProb" -> 0.25|>}
9  Out = {0.000341, <|"ZeroProb" -> 0.5, "OneProb" -> 0.5|>}
```

Table 3.1: Computing UBPD for system shown in Figure 3.4. **Lines 1, 2:** Definition of the system in Figure 3.4-A by adjacency matrix (line 1) and dynamics (line 2). **Lines 3-5:** Calculation of individual probabilities that each node of the system will take values 0/1 across the whole output repertoire. **Lines 7-9:** time of computation in seconds and UBPD distribution.

In the IIT approach, the system is perturbed with all possible inputs to obtain the full output repertoire (Figure 3.4-C). Then, in the context of ϕ_K, UBPD corresponds to the distribution of probabilities that each node will take values 0/1 in the output/input repertoires after the perturbation. For instance, in Figure 3.5-A, full input and output repertoires are shown for network in Figure 3.4-A. Now, let's say we want to compute the future probability distribution, that is, the probability necessary to compute effect information according to (17). In this case we take output repertoire as a reference and we compute the probability of nodes in the future (outputs) taking the values 0 or 1. For node A, for example, the probability that node A takes the value of 1 is 0.25, that is 2/8, and that it takes the value of 0 is 0.75 or 6/8. These values are called the UBPD for node A. A resume of UBPD for all nodes is given in Figure 3.5-B.

Once UBPD is computed for a subject partition, in this case the full system's probability distribution is computed by multiplying UBDPs. Let's take as an example the future probability of input {0, 0, 0}, computed as $P(A) = 0 * P(B) = 0 * P(C) = 0$, that is, 0.25 * 0.75 * 0.5 (see first row in Figure 3.5-C). When all future probabilities are computed in this manner, the result is the distribution shown in Figure 3.5-D, which is exactly the same one computed in (17), as shown in Figure 3.5-E.

In general, UBDP is used to compute probability distributions of a system in the context of ϕ_K, which mirrors the "constrained/unconstrained probability distributions" in (17), that is, probability distributions of input/output patterns for specific configurations (partitions) of the system, in contrast to what IIT 3.0 does. In this last case, Mayner shows how probability distributions are computed in the context of IIT in his S1 text mentioned in (16), using terms such as "marginalization" and "virtual elements" that seem to be highly complex methods.

Then, in the context of ϕ_K, UBDP aims to obtain the same results in terms of probability distributions, in a manner equivalent to IIT but by following a different conceptual approach. Our measure ϕ_K uses adapted methods, having algorithmic complexity as a background, to compute information integration.

In Table 3.1, lines 3-5 shows Mathematica code that computes UBDP for the system specified in lines 1 and 2, that is, by means of an adjacency matrix and an array of

computations that nodes perform, or the system dynamics. Table 3.1 also shows results of this computation in this order: 1) time needed to compute, followed by probability that a node take the value zero (zeroProb) or the value one(oneProb). One can see how the results in Table 3.1 correspond to UBDP values shown in Figure 3.5-B.

We should note that for ϕ_K, computation of probability distributions seems to be a task of counting, which for huge systems would be extremely difficult or even impossible, if attempted in a classical/brute force way. But, two important facts should be pointed out here: 1) In the context of ϕ_K, UBPD is not calculated in this traditional way, but is calculated using the simulation of the behaviour of a system represented by a set of simple rules. Then for ϕ_K, an exhaustive review of repertoires is not needed to compute the individual probabilities shown in Table 1, and 2) despite strong theoretical and methodological differences between them, ϕ_K and ϕ lead to the same results.

In the next sections we derive simple rules of a system, using the perturbation test and its application to implement ϕ_K.

Chapter 4

Results

4.0.1 Compression sensitivity as informative of integration

To understand the relationship between IIT and algorithmic complexity, we shall briefly move away from the case of networks and focus on binary files and the binary programs that may generate them, the programs that are natural computable candidate models explaining the data. To illustrate the connection, let us take some extreme cases. Let's say we have a random file:

```
1 In: = randomfile=RandomChoice[CharacterRange["a", "z"],100]
2 Out = {l,q,l,d,d,x,f,e,u,l,u,b,m,y,a,l,b,l,v,a,v,v,t,l,h,o,x,d,i,a,z,t,z,b,r,y,v,h,
    c,n,s,u,l,u,w,w,e,a,i,h,h,w,f,d,d,k,o,c,a,k,u,x,v,n,v,e,c,r,t,c,g,s,g,x,y,t,c,h,k
    ,w,c,t,y,u,e,k,y,v,a,h,t,k,f,y,c,r,b,y,y,x}
3
4 In: = Compress[randomfile]
5 Out = 1:eJx1kEEOgyAQRWnSi/Q+XbUnEAUhtJoKInp6zbB5CenmZTL8mf+Hh55fdlBKxfuFp4/pfbuKj/C
    HegCL0AqNcIWy1lr4Fe7CDhqNOuM1g0zihDPcaxKP2QNTB1wWZMjY1gsnYfxzxQYaeHnscdDY5q8CkvfY
    EOBSkG1CbTC14LraGZF8xJ69UTo4bugn6Fc4hubHOuxJ0FgomVOjX1lOUdJPvg==}
6
7 In: = Length[Characters@Compress[randomfile]]
8 Out = 222
```

Table 4.1: Code in Mathematica for generation of a file with random alphabetic content and measurement of its compressed version. **Line 1** generates a file with random selection of alphabetic characters. **Line 2** compresses the file generated.

So, using the **Compress** algorithm, the resulting compressed object is even longer, this is because the compression algorithms inserts the decompression instructions together with the checksum which ends up increasing the size of the resulting object if the object was not long and compressible enough to begin with.

This is what happens if we perform a couple of random perturbations to the uncompressed file like replace the 5th and 12th characters with the letters 'k' and 'x:

```
In: = mutatedfile=ReplacePart[randomfile,{5->"k", 12->"x"}]
Out = {l,q,l,d,k,x,f,e,u,l,u,x,m,y,a,l,b,l,v,a,v,v,t,l,h,o,x,d,i,a,z,t,z,b,r,y,v,h,
  c,n,s,u,l,u,w,w,e,a,i,h,h,w,f,d,d,k,o,c,a,k,u,x,v,n,v,e,c,r,t,c,g,s,g,x,y,t,c,h,k
  ,w,c,t,y,u,e,k,y,v,a,h,t,k,f,y,c,r,b,y,y,x}
```

Table 4.2: Code in Mathematica for replacing 5th and 12th characters by 'k' and 'x' respectively.

The difference between the original and perturbed files is:

```
In := SequenceAlignment[randomfile,mutatedfile]//Column
Out= {l,q,l,d} {{d},{k}} {x,f,e,u,l,u} {{b},{x}} {m,y,a,l,b,l,v,a,v,v,t,l,h,o,x,d,i
  ,a,z,t,z,b,r,y,v,h,c,n,s,u,l,u,w,w,e,a,i,h,h,w,f,d,d,k,o,c,a,k,u,x,v,n,v,e,c,r,t,
  c,g,s,g,x,y,t,c,h,k,w,c,t,y,u,e,k,y,v,a,h,t,k,f,y,c,r,b,y,y,x}
```

Table 4.3: SequenceAlignment function to compare two sequences, randomfile and mutatedfile, identifying similarities and differences between them. The //Column operator formats the output into a vertical column for clarity. The output displays aligned segments: identical portions (e.g., {l,q,l,d}, {x,f,e,u,l,u}) are shown alongside divergent segments (e.g., {{d},{k}}, {{b},{x}}), indicating where mutations or differences occur in the sequences

The files only differ by 2 characters, which can be counted using the following code:

```
In := Total[Length /@
First /@ Select[SequenceAlignment[randomfile, mutatedfile], Head[#[[1]]] == List&]]

Out = 2
```

Table 4.4: Calculation of the total length of aligned sequence segments from the comparison of *randomfile* and *mutatedfile* using **SequenceAlignment**. The Select function filters segments where the head of the first element is a List, ensuring only structured alignments are considered. The **Length/@First** computes the length of the first element in each selected segment, and **Total** sums these lengths. This metric quantifies the extent of aligned regions, useful for analysing sequence similarities in bioinformatics or data comparison studies.

That is, 2/100 or 0.02 percent.

On the other hand, let's take a simple object consisting of the repetition of a single object, say the letter e:

```
In := simplefile= Table["e",100]
Out = {e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,
  e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e
  ,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e}
```

Table 4.5: Generation a sequence named *simplefile* consisting of 100 identical characters, specifically the letter "e".

A shortest program to generate such a file is just:

```
Table["e", 100]
```

Table 4.6: Short probram for generated sequence of 100 characters, "e".

In other languages this could be produced by an equivalent **'For'** or **'Do-While'** program. We can now perturb the program again, without loss of generality. Let's allow the same 2 perturbations to the data only, and not to the program instructions (we will cover this case later). The only places that can be modified are thus 'e' or 1 instead of 5, say: **Table["a",500]**

```
In := Length/@First/@Select[SequenceAlignment[Table["a",500],Table["e",100]],Head
  [#[[1]]]==List&]
Out = {500}
```

Table 4.7: Calculation of similarity or divergence between two uniform sequences: one comprising 500 instances of the letter "a" and another with 100 instances of the letter "e", generated using the **Table** function. The **SequenceAlignment** function aligns these sequences, and **Select** filters segments where the head of the first element is a **List**, ensuring only structured alignments are considered. The **Length**/**@First** computes the length of the first element in each selected segment, yielding a list of lengths suitable for analysing sequence similarity or divergence.

Now, the original and decompressed versions differ by 500 elements, and not just a small fraction (compared to the total program length) as in the random case. This will happen in the general case with random and simple files; random perturbations will have a very different effect on each case.

An object that is highly integrated among its parts means that one can explain or describe part of each part with some other part when the object is algorithmically simple; then these parts can be compressed by exploiting the information that the said other parts carry over from yet others, and the resulting program will be highly integrated only if the removal of any of these parts has a non-linear effect on its generating program. In a random system, no part contains any information about any other, and the distribution of the individual algorithmic-content contribution of each element is a normal distribution around the mean of the algorithmic-content contributions, hence poorly integrated and trivial. So integrated information is a measure of sophistication, filtering out simple and random systems, and only ascribing high algorithmic information content to highly integrated information systems.

The algorithmic information calculus thus consists of a 2-step procedure to determine:

1. The complexity of the object (e.g. string, file, image)

2. The elements in that object that are less, more, or not sensitive to perturbations that can 'causally steer the system,' i.e. causally modify an object in a surgical algorithmic fashion rather than on the basis of guesswork based on statistics.

Note that this causal calculus is semi-computable, and one can perform guiding perturbations based upon approximations (15, 44). Also note that we did not cover the case in which the actual instructions of the program were perturbed. This is actually just a subcase of the previous case, that separates data from program. For any program and data, however, we conceive an equivalent Turing machine with empty input, thus effectively embedding the data as part of its instructions. Nevertheless, the chances of modifying the instruction **Print[]** in the random file case are constant, and for the specific example are: $7/107 = 0.0654$. While for the non-random case, the probability of modifying any piece of the **Table[]** function is: $8/12 = 0.666667$. Thus, the break-up of a program of a highly causally generated system is more likely under random perturbations.

Notice similarities to a checksum for, e.g., file exchange verification (e.g. from corruption or virus infection for downloading from the Internet), where the data to be transmitted is a program and the data block to check is the program's output file (which acts as a hash function).

Unlike regular checksums, the data block to check is longer than the program, and the checking is not for cryptographic purposes. Moreover, the dissimilarity distance between the original block (shared information) and the output of the actual shared program provides a measure of both how much the program was perturbed and the random or nonrandom nature of the data compressed by the program. And just like checksums, one cannot tamper with the program without perturbing the block to be verified (its output), without significantly changing the output (block) if what the program has encoded is nonrandom and therefore causally/recursively/algorithmically generated. Of course all the theory is defined in terms of binary objects, but for purposes of illustration and with no loss of generality we have shown actual programs operating on larger alphabets (ASCII). And we also decided to perform perturbations on what seems to be the program data and not the program itself (though we have seen that this distinction is not essential) for illustration purposes, to avoid the worst case in which the actual computer program becomes non-functional.

Yet, this means that the algorithmic calculus is actually more relevant, because it can tell us which elements in the program break it completely and which ones do not. But what happens when changes are made to the program output and not the program instructions? Say we exchange an arbitrary e for an a in our simple sequence consisting of a single letter, e.g. the third entry ('a' for 'e'):

If we were to look to the generating program of the perturbed sequence, this would need to account for the 'a', e.g.

```
In := ReplacePart[Table["e",100],3->"a"]
Out = {e,e,a,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,
  e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e
  ,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e}
```

Table 4.8: Creation of a sequence of 100 identical characters, all "e", using the **Table** function, and then modifies it with **ReplacePart** to substitute the character at position 3 with "a". The result is a uniform sequence with a single variation.

where the second program is longer than the original one, and has to be, if the sequence is simple, but the program remains unchanged if the file is random because the shortest program of a random sequence is the random sequence itself, and random perturbations keep the sequence random. Furthermore, every element in the simple example consisting of repetitions of 'e' has exactly the same algorithmic content contribution when changed or removed, as all programs after perturbation are of the form:

```
ReplacePart[Table["e",100],n->x]
```

Table 4.9: Code for replacement of the character at position n with 'x'

Notice also how this is related to ϕ and possibly any measure of integrated information based on the same principles.

We can now apply all these ideas to the language of networks, with respect to which IIT has, for the most part, been defined. We have shown before that networks with different topologies have different algorithmic complexity values (47), in accordance with the theoretical expectation. In this way, random ER graphs, for example, display the highest values, while highly regular and recursive graphs display the lowest (43). Some more probabilistic, but yet recursively generated graphs are located between these 2 extremes (41). Indeed, the algorithmic complexity K of a regular graph grows by $O(logN)$, where N is the number of nodes of the graph, as in a highly compressible complete graph. Conversely, in a truly random ER graph, however, K will grow by $O(\log E)$, where E is the number of edges, because the information about the location of every edge has to be specified.

In what follows we will perform some numerical tests strengthening our analytic derivations.

4.0.2 Finding simple rules in complex behaviour

A perturbation test is applied to systems which IIT is interested in. The set of answers is analyzed in order to find the rules that 1) make it possible to simulate the behaviour of the system, 2) define their computability power, that is, rules that give an account of what the system can and cannot compute, and 3) rules able to describe and predict behaviour of the same system. The following procedure was applied to estimate ϕ_K.

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

2. Results in step one were analyzed in order to reduce the dynamics of a system to a set of simple rules. That is, in keeping with the claims of natural computation, we found simple rules to describe a system's behaviour.

3. Rules found in step 2 were used to generate descriptions of what a system is or is not capable of computing and under what initial conditions, without having to calculate the whole output repertoire.

4. A combination of rules found in steps 2 and 3 was used to develop procedures for predicting the behaviour of a system, that is, whether it is possible to have reduced forms that express complex behaviour. Knowing what conditions are necessary for the system to compute something, it is possible to pinpoint where in the whole map of all possible inputs (questions) of a system such conditions may be found.

5. Once rules in steps 2 and 3 are formalized, ϕ_K was turned into a kind of interrogator whose purpose was to ask questions of a system about its own computational capabilities and behaviour.

This kind of analysis allowed us to find that the information distribution in the complex behaviour of systems analyzed in IIT followed a distribution replicated at several scales that is usually and informally identified as 'nested' or 'fractal', and means that it is susceptible of being summarized in simple rules by iteration or recursion, just as is the case with fractals proper. These properties are used to find compressed forms to express answers given by a system when asked for explanations of its own behaviour.

This means that, as noted before, ϕ_K does not compute the whole output repertoire for a system but uses simple rules to express the whole behaviour of the system. Interestingly, the way in which we proceed appears to be connected to whether or not the system itself can explain its behaviour, or rather whether it can see itself to be capable of producing its behaviour from an internal experience (configuration) which is then evaluated by an observer. So ϕ_K takes the form of an automatic interrogator that, in imitation of the perturbation test, asks questions of the form *are you capable of this specific configuration? (pattern), and if so, say where, in the map of the behavioural repertoire, I can find it.*

The benefit of representing systems using simple rules is that it allows an alternative calculation closer to algorithmic complexity and the potential to reduce the number of calculations to derive an educated estimation as compared to the original version of IIT 3.0.

At this point, it is not possible to explain how simple rules define a system in the context of ϕ_K without talking about the pattern of distribution of information in the behaviour of systems like those studied in IIT.

4.0.3 Simple rules and the pattern of distribution of information

As shown in (46), 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 (46).

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 knowing the rules implemented by each node in a system and finding the rules that describe its behaviour over time. To achieve the second, as perhaps could be done for the "whole [of] scientific practice" (46), we found it useful to perform perturbation tests in order to deduce the behaviour of the subject systems. Results were analyzed and a pattern in the distribution of information was found to characterize the behaviour of these kinds of systems. Then, as was to be expected, replicating behaviours were amenable to being expressed with simple formulae.

In order to explain how simple rules were found and implemented in ϕ_K, consider as an example the 7-node system shown in Figure 4.1 whose behaviour is computed by perturbing the system on all possible inputs. The code for the computational definition of the network is shown in Appendix in Table 4.10, while the results, or the complete output repertoire, is shown in Figure 4.2 in the Appendix.

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

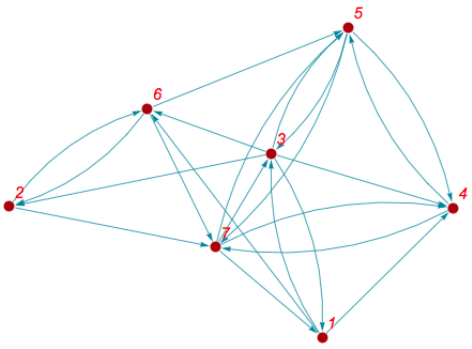

Figure 4.1: 7-node system. **Left:** Adjacency matrix. **Right:** Network representation.

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

Table 4.10: Mathematica code for computing the complete repertoire of inputs and outputs for a network, based on its dynamics. **Line 1:** Defines the adjacency matrix, where each row represents a node within the network. A value of one indicates a connection between a given node and a specific other node. **Line 2:** Specifies the dynamics, representing the function that each node executes upon receiving input from its connected nodes. **Line 3:** The 'runDynamic' function calculates the exhaustive repertoire of inputs, determined by the network's order, feeds these into the defined network, and returns the corresponding output repertoire. The output for this specific case is shown in Figure 4.2

The strategy adopted to find rules that govern a system's behaviour is the same used in almost any branch of science, which is to say we separately observe the behaviour of some of the components of a phenomenon, in this case nodes, while bearing in mind that this behaviour is not isolated but rather the by-product of interacting elements, or in other words, we observe individual behaviours without losing sight of the whole.

When we observe the behaviour shown in Figure 4.2 corresponding to the system shown in 4.1, we notice mostly chaotic behaviour but with subtle repetitions of certain patterns.

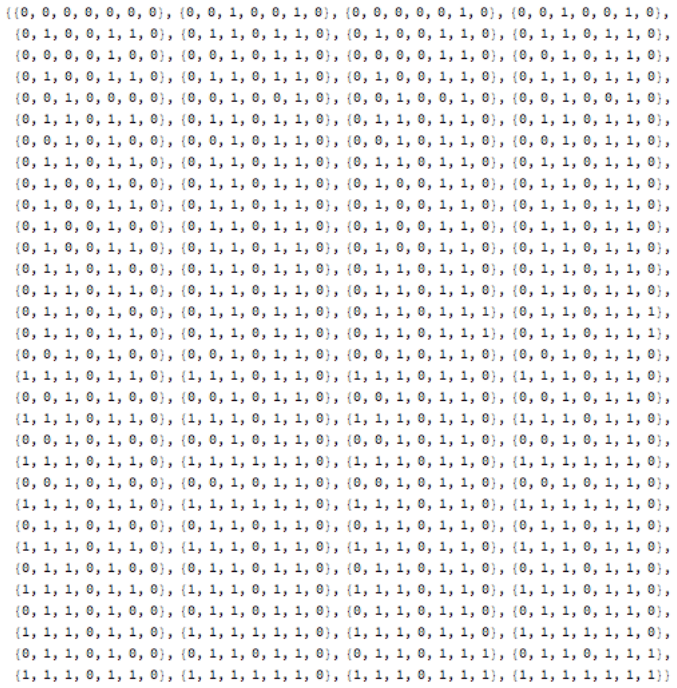

Figure 4.2: Output repertoire for 7-Node network defined in Table 4.10.

When the behaviour of elements is isolated, the picture appears clearer. For example, Tables 4.12 and 4.13 shows the isolated behaviour of nodes {4} and {5} of the same subject system as consequence of running code in Table 4.11.

```
analysis07[[All, 4]]
analysis07[[All, 5]]
```

Table 4.11: Mathematica code for isolation of behaviour of node 4 and 5 of the 7-node system shown in 4.1.

```
{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, 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}
```

Table 4.12: Isolated outputs for node 4 of the 7-node system shown in 4.1 after perturbation.

```
{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 4.13: Isolated outputs for node 5 of the 7-node system shown in 4.1 after perturbation.

Notice that now that behaviour of nodes is isolated, it is possible to expressions that describe the behaviour of the subsystems or isolated nodes.

Mathematical descriptions are shown in Figure 4.3 in what we call **Behaviour Tables**. It is important to note that these Behaviour Tables are used here just as a mathematical artifact to explain how regularities in the behaviour of the system emerge and can be explained in terms of the organisational definition of the system itsef.

In behaviour tables shown in 4.3 the lowest rows (within braces) correspond to a compressed representation of behaviours shown in Tables 4.12 and 4.13.

Let's consider first the compressed expressions of behaviour for node {4} in Behaviour Table for node {4} shown in the left side of the Figure 4.3. The compressed expression of the behaviour of the node {4} is shown in the expression 4.1:

$$85 \to 0, \{\{\{1 \to 1, 1 \to 0\}\} \to 2\}, 4 \to 0\} \to 2, 16 \to 0, \{\{\{1 \to 1, 1 \to 0\}\} \to 2, 4 \to 0\} \to 2 \tag{4.1}$$

The expression 4.1 must be readed as follow:

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 4.3: Behaviour Tables for 7-Node system shown in Figure 4.1, and defined computationally in 4.10. **Left:** Behaviour Table for node 4 that descrive output shown at 4.12. **Right:** Behaviour Table for node 5 that descrive output shown at 4.13. From left to right and up to down, **Node** column lists input-nodes that feed the target node. **node-1=pow**: Stays as mnemonic for "Content of **Node** column minus one, which determines the power to use". This column computes the power used to transform a pattern in the world of the target n-node systems from binary to decimal. **2^(pow-1)** column is the result of the binary to decimal transformation operation, and is equal to 2 powered by column **node-1=pow**. The fourth column contains divisions between indexed elements of the column **2^(pow-1)** where n is the index, whose value is equal to the element $n+1$ divided by the element indexed as n

- 85 occurrences of the digit 0, followed by
- Twist the patter 1,0, followed by 4 repetitions of digit 0, followed by
- 16 repetitions of digit 0, followed by
- Twist the patter 1,0, followed by 4 repetitions of digit 0.

Now, take in account that the purpose of Behaviour Tables and its compressed expressions are not to give a formal descritption of the isolated patters in wn in Tables 4.12 and 4.13, but to create intuition on the hightlights of our method.

Notice also that in the explanation of the compressed expression 4.1 the following list of regularities for the column **2^(pow-1)** respect to occurrences of number zero:

- 85 (the total sum of values in the column), followed by
- 1 acurrence of number zero, followed by
- 16 repetitions of digit 0, followed by
- 4 repetitions of the digit 0.

The occurrences of the digit 0 and how they appear in the patter of the expression 4.1 correspond to the regularities found in the column **2^(pow-1)** in Behaviour Table for node {4}.

Same analysis can be done easily for node {5}.

Notice also that the representation used in this isolation of behaviours is expressed in terms of the nodes that "feed" into target nodes of this example (**Node** column in

the Behaviour Tables shown in 4.3), namely nodes {4, 5} whose inputs, according to Figure 4.3 are: for node {4}: {1,3,5,7}, and for node 5: {3,4,6,7}.

This first shallow analysis works to yield the intuition that the behaviour of an isolated node can be expressed as a series of regularities in terms of its inputs. In this context, intuition tells us that the greater the number of regularities, the shorter the description; then if no patterns are detected the chances of a causal relationship are lower.

Perspective changes when rule/algorithm or compressed expression of behaviour is not constructed from regularities identified at first sight, but from intrinsic algorithmic properties. In this latter case, behaviour of systems can be expressed as patterns of information with a distribution replicable at different scales, what we here call *fractal representation* or *fractal behaviour*. To explain what we mean by fractal, we introduce characteristics of distribution of information for the 7-node system shown in Figure 4.1 analyzed using ϕ_K. This implementation is shown in Table 4.14.

```
1  In := cm07= {{0, 0, 1, 0, 0, 0, 1},
2              {0, 0, 1, 0, 0, 1, 0},
3              {1, 0, 0, 0, 1, 0, 1},
4              {1, 0, 1, 0, 1, 0, 1},
5              {0, 0, 1, 1, 0, 1, 1},
6              {1, 1, 1, 0, 0, 0, 0},
7              {0, 1, 0, 1, 1, 1, 0}};
8  In := dyn07= {"AND", "OR", "OR", "AND", "OR", "OR", "AND"};
9
10 (* Computingplacesin outputrepertoirewherenode 4 = 0 *)
11 In := res070= onPossibleBehaviour[{4}, {0}, dyn07, cm07]
12
13 (* Summarizedrepresentationoffractalbehaviour*)
14 In := gp= givePlaces[res070["DecimalRepertoire"], res070["Sumandos"]]
15
16 (* Compressedrepresentationoffractalbehaviour*)
17 Out = <|"DecimalRepertoire"-> {0, 1, 4, 5, 16, 17, 20, 21, 64, 65, 68, 69,80, 81, 8
     4}, "Sumandos"-> {0, 2, 8, 10, 32, 34, 40, 42}|>
18
19 (* Unfoldedrepresentationoffractal behaviour*)
20 Out = {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, 8
     2, 83, 84, 86, 88, 89, 90, 91, 92, 94, 96, 97, 98, 99, 100, 101, 102, 103, 104, 1
     05, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 118, 120, 121, 122, 12
     3, 124, 126}
```

Table 4.14: ϕ_K asking for accounts of information distribution in behaviour of 4th node of the system shown in Figure 4.1. **Lines 1-8**: Definition of the 7-node system by means of adjacency matrix and its internal dynamics. **Line 11:** ϕ_K's code asking for zero digit location in the whole behaviour of node 4. **Line 17:** Compressing answer given by the system in line 11. **Line 20:** unfolded answer of the system in Line 17.

Table 4.14 shows how behaviour of the system shown in Figure 4.1 can be expressed as simple rules following an analysis based on a querying scheme that results in a reduced form to express its information distribution as a pattern replicated at different scales

or as a fractal form. Answers given by systems join facts explored above on regularities and the fractal distribution of information. It is important to note that the querying scheme has to be computable and algorithmically random in order to avoid introducing an artificially random-looking behaviour from the observer (experimenter/interrogator) to the observed (the system in question).

In 4.14, after defining the target system by means of an adjacency matrix and a dynamics vector (lines 1 to 8), ϕ_K can be regarded as testing: *how 0 is distributed in node {4} in the system of seven nodes* (line 11).

The target system reacts to the ϕ_K's query and it "answers" in a compressed form (Table 4.14, line 17). The result can be represented in compressed form, expressed as a tiny rule that represents what we have called a fractal pattern. Such an expression is defined, as can be seen in the line 17 in Table 4.14, by two variables: *DecimalRepertoire* that holds points distanced in different proportions where the patterns defined by the *Sumandos* variable must be reproduced. This means that in order to unfold the whole distribution (of digit zero), the pattern of numbers in *Sumandos* must be added to each value in *DecimalRepertoire*.

Once this 'fractal' simple rule is unfolded, we obtain the ordinal places where, in the whole behaviour of node {4}, digit 0 can be found (see line 20 in 4.14). The accuracy of this answer can be verified by counting ordinals where, for node {4}, its output = 0 in Table shown in the Figure 4.2, taking into account that counting starts at 0.

In summary, ϕ_K is turned into a kind of interrogator that asks a system about its own behaviour. On the other hand, a system is an analyzer and self evaluator capable to implement the set of rules that answers in different ways, depending on the information requested. This is unlikely with traditional approaches to ϕ, whose representation of the system consists of the whole output repertoire of the system, which might represent an important disadvantage when large networks are analyzed. ϕ_K's answers use compressed forms taking advantage of the fractal distribution of the information in the behaviour of the system, for which the answering interface is a function of its input related to each node in question.

Obviously the whole behaviour of a system is not about isolated elements, but about elements interacting in a non-linear manner, as IIT 3.0 makes clear. This last, broader view is also addressed in terms of ϕ_K, and explained in the following sections. In the next one, the advantages of simple rules over classical/naive approaches based on an exhaustive calculus and review of whole repertoires held in memory will be established.

4.0.3.1 Automatic meta-perturbation test

It can be seen that this querying system is similar to the programmability tests suggested in (15, 35, 37) based on questions designed to ascertain the programmability of a dynamical system.

The last section shows that systems implemented as simple rules that give rise to complex behaviour enable the system itself to "respond" to questions about where, in the chain of digits that conform to its behaviour (of a specific node), a certain pattern

is to be found. And the fractal nature of information distribution in behaviour allows us to answer complex distribution questions in short forms. In this section, we show the advantages of using an (automatic) perturbation test based on simulation of behaviour using simple rules over the original version of IIT 3.0 based on the "bottleneck principle" (18) in computing integrated information.

Taking up the original perturbation test, questions take the form: *what is the output (answer) given this query (input)?*. But in ϕ_K, since questions look for explanations of the behaviour of the system itself, they take the form: *tell me if this pattern is reachable, and if so, tell me where, in the behavioural map, it is possible to find it.*

The following example aims to show that even is possible to implement an interrogator and its corresponding analyzer that can answer questions about the behaviour of the target system, ϕ_K surpaces the capabilities of traditional approaches as ϕ first and importantly minimizing the time and computational sources.

One naivie implementation of an analyzer for a system of 9 nodes is defined in Table 4.15 while the structure of the network is show in Figure 4.4. Table 4.15 shows this approach using filters to find patters. Lines 3-12 defines the adjacency matrix and dynamics vector of the system. Line 14 runs the internal dynamics over the exhaustive repertoire of inputs obtaining the exhaustive repertoire of outputs. Then lines 16-18 filter over all outputs cases where nodes {8, 9} = {1,1}.

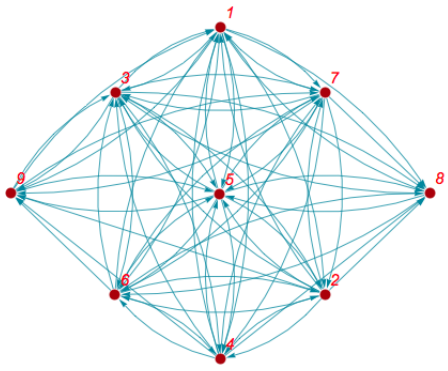

Figure 4.4: Network example with 9 nodes

```
1  mmu=MemoryInUse[];
2  AbsoluteTiming[
3  In := cmTest= {{0,1,1,1,1,1,1,1,1},
4                {1,0,1,1,1,1,1,1,1},
5                {1,1,0,1,1,1,1,1,1},
6                {1,1,1,0,1,1,1,1,1},
7                {1,1,1,1,0,1,1,1,1},
8                {1,1,1,1,1,0,1,1,1},
9                {1,1,1,1,1,1,0,1,1},
10               {1,0,1,0,1,1,0,0,0},
11               {1,0,1,0,1,0,1,0,0}};
12 In := dynTest= {"OR","XOR","AND","XOR","AND","AND","OR","OR", "AND"};
13
14 In := tdTest=runDynamic[cmTest, dynTest];
15
16 In := inTest= tdTest["RepertoireInputs"];
17 In := outTest= tdTest["RepertoireOutputs"];
18
19 (*looking forpatternwhere{8,9}={1,1}*)
20 In :=  For[i=1, i<=Length[outTest],i++,
21     If[(outTest[[i]][[8]]===1 && outTest[[i]][[9]]===1),
22         outTest[[i]]=Style[outTest[[i]],{Bold,Red}]
23     ];
24 ];
25 (*Showing results*)
26 In :=  assoc=AssociationThread[inTest, outTest]
27 ]
28 MemoryInUse[]-mmu
```

Table 4.15: Mathematica code for definition of the 9-Node system shown in Figure 4.4. **Lines 1-12:** Definition of the 9-Node system by means of adjacency matrix and its internal dynamics. **Line 14:** Instruction for runing the internal dynamics overl exhaustive input repertoire. **Lines 17, 18:**: Retrieving exhaustive repertoores of inputs and outputs. **Lines 20-24:** Filtering over all cases where {8,9}=1,1. **Lines 1, 28:** Measuring time and memory used by the program.

Results of running the code in 4.15 are shown in 4.16, where time in seconds is given, followed by the cases in repertoire of inputs and outputs where where the pattern {8,9}=1,1 is found. Finally the memory used by the program is given in bytes.

```
Out = 0.024581
Out = <|runDynamic[{{0, 1, 1, 1, 1, 1, 1, 1, 1}, {1, 0, 1, 1, 1, 1, 1, 1,
    1}, {1, 1, 0, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 0, 1, 1, 1, 1,
    1}, {1, 1, 1, 1, 0, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 0, 1, 1,
    1}, {1, 1, 1, 1, 1, 1, 0, 1, 1}, {1, 0, 1, 0, 1, 1, 0, 0,
    0}, {1, 0, 1, 0, 1, 0, 1, 0, 0}}, {"OR", "XOR", "AND", "XOR",
    "AND", "AND", "OR", "OR", "AND"}]["RepertoireInputs"] ->
Out = runDynamic[{{0, 1, 1, 1, 1, 1, 1, 1, 1}, {1, 0, 1, 1, 1, 1, 1, 1,
    1}, {1, 1, 0, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 0, 1, 1, 1, 1,
    1}, {1, 1, 1, 1, 0, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 0, 1, 1,
    1}, {1, 1, 1, 1, 1, 1, 0, 1, 1}, {1, 0, 1, 0, 1, 1, 0, 0,
    0}, {1, 0, 1, 0, 1, 0, 1, 0, 0}}, {"OR", "XOR", "AND", "XOR",
    "AND", "AND", "OR", "OR", "AND"}]["RepertoireOutputs"]|>}
Out = 17776
```

Table 4.16: Results of running code shown in Table 4.15. This outputs are 1) execution time in seconds, 2) filtered input cases where {8,9}=1,1 is found, 3) filtered output cases where {8,9}=1,1 is found, and 4) memory used by the programin bytes

.

For comparing purposes, ono more things have to be pointed respect the results shown in Table 4.15, and this is the format of the outputs and inputs, where the rough cases are shown. This is, with no showing null analysis or highlights that give and idea how these inputs and outputs are related with the targeted nodes.

The implementation of ϕ_K for specifically turnint it into an automatic interrogator is shown in Table 4.17. Notice that this approach is also based on analyze the system networks shown in Figure 4.4. In line 1, in the list code shown in 4.17, it is possible to see how at low level of implementation ϕ_K asks questions of a system. This line should be interpreted as, *Can you compute the pattern {8,9} = {1,1} when {8,9}-> {"OR", "AND"}? If yes, tell me under what conditions you can do so.* Notice that here we are showing the low level implementation of this function. What really happens is that our high level implementation easily identiy input nodes of the targeted nodes, and its internal dynamics from the especification of the system.

```
In := combiningRepersWithSharedInputs[{1, 3, 5, 6}, "OR", 1, {1, 5, 7, 3}, "AND", 1
  ]

Out = 0.000736
Out = <|"Combination" -> {{1, 0, 0, 0, 0}, {1, 0, 0, 0, 1}, {0, 1, 0, 0,
    0}, {0, 1, 0, 0, 1}, {1, 1, 0, 0, 0}, {1, 1, 0, 0, 1}, {0, 0, 1,
    0, 0}, {0, 0, 1, 0, 1}, {1, 0, 1, 0, 0}, {1, 0, 1, 0, 1}, {0, 1,
    1, 0, 0}, {0, 1, 1, 0, 1}, {1, 1, 1, 0, 0}, {1, 1, 1, 0, 1}, {0,
    0, 0, 1, 0}, {0, 0, 0, 1, 1}, {1, 0, 0, 1, 0}, {1, 0, 0, 1,
    1}, {0, 1, 0, 1, 0}, {0, 1, 0, 1, 1}, {1, 1, 0, 1, 0}, {1, 1, 0,
    1, 1}, {0, 0, 1, 1, 0}, {0, 0, 1, 1, 1}, {1, 0, 1, 1, 0}, {1, 0,
    1, 1, 1}, {0, 1, 1, 1, 0}, {0, 1, 1, 1, 1}, {1, 1, 1, 1, 0}, {1,
    1, 1, 1, 1}},
"Filtered" -> {{1, 1, 1, 0, 1}, {1, 1, 1, 1, 1}},
"jn" -> {1, 3, 5, 6, 7},
"nn" -> {1, 2, 3, 4, 5},
"DecRep" -> {85, 117}|>

Out = 2440
```

Table 4.17: ϕ_K algorithmic querying of the system about its own behaviour as shown in Figure 4.4. **Line 1**: Query: Is it possible for this system to compute *{8,9} = {1,1} when {8,9}-> {"OR", "AND"} and whose input nodes are {1,3,5,6} and {1,5,7,3} respectively?*. The results show that the system does compute 1) time of computation in seconds, 2) possible candidates for combination of conditions, 3) The especific input patters for specific notes ("Filtered" and "jn" keys), this is when {1,3,5,6,7} = {{1,1,1,0,1},{1,1,1,1,1}} 4) Decimal representations of summandos to use for unfolding the complete dynamics of the system and 5) the memory used in bytes.

In this example, in the first place ϕ_K tries to find conditions needed to compute a specific output. As Table 4.17 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}.

The reader would note here that the answers offered by the system actually are conditions or inputs needed by the system to compute specific input in a format equivalent to Holland's schemas. The schemas' equivalent form for this case would be: {{1,*,1,*,1,0,1,*,*}, {1,*,1,*,1,1,1,*,*}}, where '*' is a wildcard that means 0/1 (any symbol). Such schemas correspond exactly to the generalized answer offered by the system, that is: {1,3,5,6,7} = {{1,1,1,0,1},{1,1,1,1,1}}.

This answer, like the Holland's schema theorem (11), works by imitating genetics, where a set of genes are responsible for specific features in phenotypes. What ϕ_K retrieves is the general information that yields specific inputs for the current system.

Probably the greatest advantage of the approach using ϕ_K in querying samples has to do with the computation time needed to retrieve such information, as compared with a traditional/naivie (Table 4.16) approach: 1/10 in the case shown just right above. But this effects are magnified when the number of nodes increases, this due the fractal distriution of the information. This is causede by the continous addition of nodes into a particular analysis, since is possible to know that, if a system is strongly integrated, most of the parts of the system definition and its behaviour is involved in generating

patters and answers about itself. This is shown in Table 4.18, that shows the definition and results of runing ϕ_K quering on a network of 16 nodes, a size untractable but the implementation of IIT 3.0. In the results shown in Table 4.18 the calculation amount of time and memory required for computation of answers are shown in lines 26 and 27, respectively.

```
1  mmu=MemoryInUse[];
2  AbsoluteTiming[
3  cm16 = {{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0},
4         {0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1},
5         {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
6         {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0},
7         {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0},
8         {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
9         {0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1},
10        {1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0},
11        {0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0},
12        {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0},
13        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
14        {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0},
15        {0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0},
16        {0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
17        {1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
18        {0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0}};
19
20 dyn16 = {"AND", "OR", "OR", "AND", "OR", "OR", "AND", "OR", "OR", "AND", "OR", "OR"
     , "AND", "AND", "OR", "AND"};
21 res = onPossibleBehaviour[{5, 7, 9, 10}, {0, 0, 1, 0}, dyn16, cm16];
22 givePlaces[res["DecimalRepertoire"], res["Sumandos"]];]
23 ]
24 MemoryInUse[]-mmu
25
26 Out = {0.000011, {7.*10^-6, Null}}
27 Out = 5888
```

Table 4.18: Definition of a network of 16 nodes and its results of running ϕ_K quering. Results shown the effect of the fractal distribution and the level of integration in a network

This last suffices as proof that compression and generalization of systems in the form of simple rules based on naturally fractal information distribution has advantages over common sense or classical approaches to the analysis of complex systems, particularly in terms of the computational resources needed to compute integrated information.

All the above were applied to analyzing isolated or very simple cases. In the next section the generalization of n nodes of the system is addressed, and how this works to compute integrated information according to IIT.

4.0.3.2 Shrinking after dividing to rule

In previous sections it was shown how ϕ_K, applying a perturbation test, can deduce, firstly, what a system is capable of computing and the conditions under which a computation could be performed, and secondly, that by means of simple rules specifying a

system it is possible to obtain descriptions of its behaviour 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 description of the behaviour of a system is to know how many times specific patterns appear in whole repertoires, and thus to construct probability distributions without need of exorbitant computational resources, since these probability distributions are a key piece used by IIT to compute integrated information.

ϕ_K addressed such challenges using a two-pronged strategy consisting firstly of parallelizing the analytical process– which is no 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 of the partition of the target sets. This latter part of ϕ_K's strategy consists of two parts: 1) given a target set to be analyzed, to divide this into parts to be interrogated by ϕ_K via the implementation of an automatic test, and 2) to find the MIP or the Maximal Information Partition using the algorithm proposed and proved by Oizumi in (13).

In the context of ϕ_K, when a partition of a subject system is being analyzed, the search space for the remaining parts is significantly reduced, facilitating and acelerating the analysis of the remaining parts.

In order to illustrate this idea, take for example the code shown in Table 4.19 and its restuls in Table 4.20.

```
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
```

Table 4.19: Definition of a network of 7 nodes and consecutive application of ϕ_K quering at different levels of complexity.

```
1 {0.00076,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
2 {0.000647,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
3 {0.0009,<|"DecimalRepertoire"->{0},"Sumandos"->{0,2,8,10}|>}
4 {0.000656,<|"DecimalRepertoire"->{0,16},"Sumandos"->{0}|>}
5 {0.001248,<|"DecimalRepertoire"->{0},"Sumandos"->{0}|>}
```

Table 4.20: Comparing processing time when a system is divided to compute outputs. **Line 10-14:** ϕ_K asking the system defined in lines 1-9 for patterns filled with zeros with different lengths (3, 4 and 7) and combinations. **Lines 1-5** show, time in seconds taken for computations and answers in terms of indexes using compressed notation. In first data of this results square it can be observed that the larger the node wanted, the greater the amount of time required to perform the computation, while the time ratio decreases.

Table 4.19 shows the definition of a system of 7 nodes (lines 1-9), where a set of a progressively growing length is searched (lines 10-14). In this example ϕ_K repeatedly asks the system if it is capable of finding a growing pattern of zeros. If it is, the system is requested to show where it is possible to find the desired pattern. Obviously, larger patterns need more computations, but as can be seen in Table 4.20, in the results square, the time used by ϕ_K increases as the pattern's length increases, but it grows linearly in contrast to IIT 3.0, where it grows exponentially.

Chapter 5

Integration in Complex Networks Using Algorithmic Complexity

5.1 Metacompression

In the study of complex systems, measuring integration—the extent to which components of a system work collectively to form a unified whole—is a fundamental challenge. Integrated Information Theory (IIT), proposed by Tononi, quantifies integration through ϕ, which measures the information generated by a system beyond the sum of its parts by identifying the Minimum Information Partition (MIP). However, IIT's computational complexity grows exponentially with system size (2^N for N nodes), rendering it infeasible for large-scale networks. Inspired by Zenil's work (46), which explores algorithmic complexity as a tool for analysing network dynamics, we introduce a novel measure of integration, ϕ_K, that leverages the intrinsic spreading of information within a fractal dynamic, reflected in attractors. These attractors represent the natural convergence points of information flow, as defined by the inherent structure of a system or network. By employing concise programs derived from querying activities, the fractal distribution of information and algorithmic complexity can be utilised to efficiently capture the essence of integration..

Our approach is a pattern-based perturbation approach, that inpired on natural distribution of infomation expressed as fractal distribution that evolved to an attractor-based method, and culminated in a compressed, rule-based formulation that avoids IIT's exhaustive partitioning. This section details the conceptual evolution, mathematical formulations, and implementation in Mathematica, culminating in a scalable method to measure integration in networks of varying sizes and topologies.

5.1.1 Step 1: Initial Pattern-Based Approach

We start with a network of N nodes and E edges, aiming to measure integration by perturbing the network and observing its response. For a 9-node network with 20 edges, we initially proposed querying the network's ability to produce all $2^N = 2^9 = 512$ possible binary output patterns (e.g., $\{1, 2\} = \{0, 0\}$, $\{3, 4, 5\} = \{1, 0, 1\}$) (see Table A.1). For each pattern, we would:

1. Compute the rule (program) needed to generate the pattern, measuring its complexity $K(s_0)$ in bytes (see bellow).
2. Remove one edge, recompute the rule, and measure the new complexity $K(s_1)$.
3. Calculate the sensitivity as $\Delta K = |K(s_0) - K(s_1)|$.

For each edge ($E = 20$) and pattern ($2^9 = 512$), this required $20 \times 512 = 10,240$ simulations per network, totaling 1,024,000 for 100 networks. While more efficient than IIT's 2^{N-1} bipartitions (e.g., $2^8 = 256$ for 9 nodes), this was still computationally expensive. We also considered partitioning the network (like IIT's MIP), but this would replicate IIT's exponential cost (B_N, the Bell number), defeating our goal of efficiency.

To reduce costs, we proposed a hybrid approach:

- Select a subset of patterns (e.g., 10 subset patterns + 1 whole-system pattern, such as $\{1, 2\} = \{0, 0\}$, $\{1, 2, 3, 4, 5, 6, 7, 8, 9\} = \{0, 1, 0, 1, 0, 1, 0, 1, 0\}$).
- Compute ΔK for each edge and pattern, averaging to get ϕ_K:

$$\Delta K_i = \frac{1}{P} \sum_{j=1}^{P} \Delta K_{i,j}, \quad \phi_K = \frac{1}{E} \sum_{i=1}^{E} \Delta K_i$$

  where $P = 11$ patterns, $E = 20$ edges. This reduced simulations to $20 \times 11 = 220$ per network, or 22,000 for 100 networks—a significant improvement.

However, the pattern selection was arbitrary, and the approach still required multiple queries, potentially missing the network's intrinsic dynamics.

5.1.2 Step 2: Shifting to Attractors

Recognizing the inefficiency of querying all 2^N patterns, we shifted to a more natural approach: using the network's attractors—the set of stable states (fixed points or cycles) under its dynamics (e.g., Boolean rules like AND, OR). Attractors represent the system's possible outputs without exhaustively testing all input-output pairs, using minimal descriptions.

For a 5-node network defined by:

- Adjacency matrix: $\text{cm05} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$
- Dynamics: dyn05 = {"AND","AND","OR","AND","OR"}

Tononi's approach yields $2^5 = 32$ output patterns, many redundant (e.g., $\{0, 0, 0, 0, 0\}$ at positions 0, 16). Our attractor method, using algorithmic complexity principles, compressed this to 8 unique attractors:

$$\text{AttractorsByPosition} = \langle |0 \to \{0, 16\}, 4 \to \{1, 17\}, 16 \to \{2, 4, 6, 18, 20, 22\}, \ldots, 31 \to \{31\}| \rangle$$

where keys are decimal representations of binary states (e.g., $0 = \{0, 0, 0, 0, 0\}$), and values are positions in the full repertoire.

This reduced the problem space from 32 to 8 queries per perturbation. We proposed:

1. Compute the baseline attractor map, measure its complexity $K(s_0)$ (e.g., using ByteCount in Mathematica).
2. Perturb each edge, recompute attractors, and measure $K(s_1)$.
3. Compute $\Delta K_i = |K(s_0) - K(s_1)|$ and $\Delta A_i = |A_0 - A_1|$ (change in attractor count).
4. Define sensitivity per edge as:

$$\text{Sensitivity}_i = \Delta K_i \cdot \log_2(\Delta A_i + 1)$$

   where $\log_2$ scales the attractor count change informationally, and $+1$ ensures the term is defined when $\Delta A_i = 0$.

5. Compute ϕ_K as the average sensitivity:

$$\phi_K = \frac{1}{E} \sum_{i=1}^{E} \text{Sensitivity}_i$$

For 5 nodes and 11 edges, this required 1 baseline + 11 perturbations = 12 simulations per network, or 1,920 for 160 networks—a massive improvement over 10,240.

5.1.3 Why Attractors?

- They capture the network's intrinsic dynamics, avoiding arbitrary pattern queries.

- Perturbing edges tests integration causally: high integration means large ΔK and ΔA (attractors collapse), while low integration means small changes (attractors persist).
- No need for IIT's partitions—edge perturbations act as localized cuts, approximating the MIP's effect dynamically.

5.1.4 Step 3: Rule-Based Refinement

The attractor map provided a global view, but we noticed that each attractor could be further compressed into a rule describing its positions in the repertoire—a meta-compression explained as a fractal distribution of the information summarized as DecimalRepertoire and Sumandos (Table 4.14). For the attractor $\{0, 0, 0, 0, 0\}$:

$$\text{Rule} = \langle|\text{“DecimalRepertoire”} \to \{0, 16\}, \text{“Sumandos”} \to \{0\}|\rangle$$

This rule encodes how to generate the attractor's positions, offering a finer measure of dynamic sensitivity.

We refined ϕ_K by:

1. Computing rules for each attractor before and after perturbation.
2. Measuring rule complexity K_{rule} (via ByteCount), averaging across attractors:

$$K_{\text{rule,avg}} = \frac{1}{A} \sum_{a=1}^{A} K_{\text{rule}}(a)$$

3. Calculating the rule sensitivity:

$$\Delta K_{\text{rule,avg}} = |K_{\text{rule,avg}}(s_0) - K_{\text{rule,avg}}(s_1)|$$

4. Combining with the map sensitivity using a weight $w = 1$:

$$\phi_K = \left(\frac{1}{E} \sum_{i=1}^{E} \text{Sensitivity}_i \right) + w \cdot \Delta K_{\text{rule,avg}}$$

This dual approach—global (map) and local (rules)—captures integration at multiple scales, enhancing ϕ_K's sensitivity to subtle dynamic changes.

5.1.5 Implementation in Mathematica

We implemented this approach for some networks of variable architecture and dynamics, generating n networks per size and type (Barabási-Albert, Watts-Strogatz, Ring, Complete). The detailed algorithm can be seen at algorithm 12.

A high level run can be see as follows:

1. **Network Generation**:
   - For each node size N (min to max), generate E edges randomly.
   - Create graphs using:
     - Barabási-Albert (scale-free)
     - Watts-Strogatz (small-world)
     - Ring
     - Complete
   - Extract adjacency matrix
2. **Dynamics**:
   - Assign random Boolean rules
3. **Full Repertoire (for PyPhi)**:
   - Compute all 2^N output patterns
4. **Random State**:
   - Generate a random input and output
5. **Attractors**:
   - Compute attractors
   - Measure complexity
6. **Perturbations**:
   - For each edge, set adjMat[i,j] = 0, recompute attractors, and calculate:

     $$\Delta K_i = |K(s_0) - K(s_1)|, \quad \Delta A_i = |A_0 - A_1|, \quad \text{Sensitivity}_i = \Delta K_i \cdot \log_2(\Delta A_i + 1)$$

7. **Rule Refinement**:
   - Compute rules
   - Measure $\Delta K_{\text{rule,avg}}$ and add to ϕ_K with $w = 1$.
8. **Save Data**:

- Export to `network_data.csv` with NumPy-compatible formatting for PyPhi comparison.

Scalability: For 5 nodes (11 edges), 12 simulations per network; for 1000 networks, $\sim 12,000$ simulations—orders of magnitude fewer than IIT's 2^{N-1} bipartitions.

5.1.6 insights on Mathematica code

- **Attractors Over Patterns**: Attractors reflect intrinsic dynamics, reducing queries from 2^N to the number of unique stable states.
- **Algorithmic Complexity**: Using ByteCount as a proxy for K aligns with Zenil's Algorithmic Complexity but considering compression sensitivity, capturing rule complexity changes.
- **No Partitions**: Edge perturbations approximate MIP dynamically, avoiding exponential costs.
- **Rule Refinement**: Adds local sensitivity, making ϕ_K more robust.
- **Scalability**: Linear in edges, not exponential in nodes, unlike IIT.

Formulas Recap:

- Sensitivity per edge: $\text{Sensitivity}_i = \Delta K_i \cdot \log_2(\Delta A_i + 1)$
- Initial ϕ_K: $\phi_K = \frac{1}{E} \sum_{i=1}^{E} \text{Sensitivity}_i$
- Refined ϕ_K: $\phi_K = \left(\frac{1}{E} \sum_{i=1}^{E} \text{Sensitivity}_i \right) + w \cdot \Delta K_{\text{rule,avg}}$, $w = 1$

Overview of implementation

Our ϕ_K offers a computationally efficient alternative to IIT's ϕ, capturing integration through attractor dynamics and algorithmic complexity.

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

In order to overcome this limitation, ϕ_K attempted to find rules that not only give an account of the computability capabilities of a system, but also describe its own behaviour. In other words, we wanted to know about possibilities for finding "shortcuts to express the behaviour" of a whole system.

One other obvious limitation inherited from computability and algorithmic complexity is that of the semi-computability of the process of trying to find simple representations of behaviour. However, we are not required to find the shortest (simplest)

one but simply a set of possible short (simple) ones, which would be an indication of the kind of system we are dealing with. While one can find shorter descriptions using popular lossless compression algorithms, the more powerful the algorithms to find shortcuts and fractal descriptions, the faster the computation and the more telling the results, something that is to be expected for a relationship between the way in which integrated information is estimated, on the one hand, and algorithmic complexity.

5.2 Demonstration of Integration Measurement via ϕ_K

To justify the effectiveness of the ϕ_K approach, we demonstrate its application on a specific 5-node network, defined by its adjacency matrix and dynamics. This step-by-step example illustrates how integration is measured through perturbation, a ttractor dynamics, and algorithmic complexity, showcasing the computational efficiency and interpretability of our method compared to traditional approaches like IIT.

5.2.1 Step 0: Network Setup

We consider a 5-node network with the following parameters:

- **Adjacency Matrix**: A complete graph (with self-loops removed), represented as:

$$\text{adjMat} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

  This matrix indicates 10 edges (since it is symmetric and the diagonal is 0), reflecting a highly connected network expected to exhibit significant integration.

- **Dynamics**: Each node is assigned a Boolean rule:

$$\text{dyn} = \{\text{“OR”}, \text{“AND”}, \text{“OR”}, \text{“AND”}, \text{“OR”}\}$$

  These rules govern how nodes update their states based on inputs from connected nodes, introducing variability in the network's behavior.

5.2.2 Step 1: Compute Baseline Attractors

First, we compute the network's baseline attractors—the stable states or cycles under the given dynamics. The attractor map is:

$$\begin{aligned}\text{Baseline Attractors} = \langle|\{0,0,0,0,0\} \to \{0\}, \{0,0,1,0,1\} \to \{1\}, \{1,0,0,0,1\} \to \{4\},\\ \{1,0,1,0,0\} \to \{16\}, \{1,0,1,0,1\} \to \{2,3,5,6,7,8,9,10,11,12,13,14,\\ 15,17,18,19,20,21,22,24,25,26,27,28,30\}, \{1,0,1,1,1\} \to \{23\},\\ \{1,1,1,0,1\} \to \{29\}, \{1,1,1,1,1\} \to \{31\}|\rangle\end{aligned}$$

Here, each key (e.g., $\{0,0,0,0,0\}$) represents a binary state (converted from decimal positions in the repertoire), and the values (e.g., $\{0\}$) indicate positions in the full $2^5 = 32$ state space where this state appears as a stable attractor. Notably, the state $\{1,0,1,0,1\}$ dominates, appearing at 21 positions, suggesting a strong attractor basin.

We measure:

- **Baseline Complexity (K_0)**: $K_0 = 3160$ bytes, computed using ByteCount in Mathematica, serving as a proxy for Kolmogorov complexity (46).
- **Baseline Attractor Count (A_0)**: $A_0 = 8$, the number of unique attractors, reflecting the network's dynamic diversity.

5.2.2.1 Step 2: Count Edges

The network has 10 edges (counted as 1s in the upper triangular part of the adjacency matrix, excluding the diagonal):

$$\text{Number of Edges} = 10$$

This determines the number of perturbations we will perform.

5.2.2.2 Step 3: Perturb Edges and Compute Sensitivities

We perturb each edge by setting its connectivity to 0 and recompute the attractors, measuring the sensitivity to each perturbation. The process is illustrated for select edges:

- **Perturbation of Edge (1,2)**:

$$\text{Perturbed Adjacency Matrix} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{aligned} \text{Perturbed Attractors} = \langle | &\{0,0,0,0,0\} \to \{0\}, \{0,0,1,0,1\} \to \{1,2,3\}, \{1,0,0,0,1\} \to \{4\}, \\ &\{1,0,1,0,0\} \to \{16\}, \{1,0,1,0,1\} \to \{5,6,7,8,9,10,11,12,13,14,15, \\ &17,18,19,20,21,22,24,25,26,27,28,30\}, \{1,0,1,1,1\} \to \{23\}, \\ &\{1,1,1,0,1\} \to \{29\}, \{1,1,1,1,1\} \to \{31\} | \rangle \end{aligned}$$

$$K_1 = 3176 \text{ bytes}, \quad A_1 = 8$$

$$\Delta K = |3160 - 3176| = 16, \quad \Delta A = |8 - 8| = 0$$

$$\text{Sensitivity}_{(1,2)} = \Delta K \cdot \log_2(\Delta A + 1) = 16 \cdot \log_2(0 + 1) = 16 \cdot 0 = 0$$

The zero sensitivity indicates that removing this edge does not significantly alter the network's dynamics, as both the number and complexity of attractors remain stable.

- **Perturbation of Edge (2,4)** (a more impactful example):

$$\text{Perturbed Adjacency Matrix} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{aligned} \text{Perturbed Attractors} = \langle | &\{0,0,0,0,0\} \to \{0\}, \{0,0,1,0,1\} \to \{1\}, \{1,0,0,0,1\} \to \{4\}, \\ &\{1,0,1,0,0\} \to \{16\}, \{1,0,1,0,1\} \to \{2,3,5,6,7,8,9,10,11,12,13,14, \\ &15,17,18,19,20,22,24,25,26,27,28,30\}, \{1,1,1,0,1\} \to \{21,29\}, \\ &\{1,1,1,1,1\} \to \{23,31\} | \rangle \end{aligned}$$

$$K_1 = 2808 \text{ bytes}, \quad A_1 = 7$$

$$\Delta K = |3160 - 2808| = 352, \quad \Delta A = |8 - 7| = 1$$

$$\text{Sensitivity}_{(2,4)} = \Delta K \cdot \log_2(\Delta A + 1) = 352 \cdot \log_2(1 + 1) = 352 \cdot 1 = 352$$

Here, the sensitivity is significant, reflecting a notable change in the attractor map: the number of attractors decreases from 8 to 7, and the complexity drops by 352 bytes, indicating that edge (2,4) plays a critical role in the network's integration.

- **Remaining Edges**: Similar perturbations are performed for all 10 edges. Most edges (e.g., (1,3), (1,4), (1,5), etc.) yield $\text{Sensitivity}_i = 0$, as $\Delta A = 0$, meaning the attractor count remains stable. However, edge (4,2) also yields $\text{Sensitivity}_{(4,2)} = 352$, mirroring the impact of edge (2,4) due to the symmetry of the undirected graph.

The sensitivities across all edges are:

$$\text{Sensitivities} = \{0, 0, 0, 0, 0, 0, 352, 0, 0, 0, 0, 0, 352, 0, 0, 0, 0, 0, 0, 0\}$$

Filtering out zeros (as per our methodology), we retain:

$$\text{Sensitivities} = \{352, 352\}$$

5.2.2.3 Step 4: Compute Initial ϕ_K

The initial ϕ_K is the mean of the non-zero sensitivities:

$$\phi_K = \frac{1}{|\text{Sensitivities}|} \sum \text{Sensitivity}_i = \frac{352 + 352}{2} = 352$$

This value indicates that, on average, the network's dynamics are moderately sensitive to perturbations, with only two edges significantly affecting the attractor map.

5.2.2.4 Step 5: Compute Rules and Refine ϕ_K

To refine ϕ_K, we compute the rule-based sensitivity:

- **Baseline Rules**: For each attractor, we generate a rule encoding its positions (e.g., for $\{0, 0, 0, 0, 0\} \to \{0\}$, the rule is $\langle|\text{“DecimalRepertoire”} \to \{0\}, \text{“Sumandos”} \to \{0\}|\rangle$).

$$K_{\text{rule},0} = 585 \text{ bytes (average across 8 rules)}$$

- **Perturbed Rules**: Perturbing edge (1,2), we recompute rules for the original attractors:

$$K_{\text{rule},1} = 587 \text{ bytes}$$

$$\Delta K_{\text{rule}} = |585 - 587| = 2$$

This small ΔK_{rule} suggests that the rules are relatively stable, but it adds a local sensitivity component to our measure.

The refined ϕ_K with $w = 1$ is:

$$\phi_K = 352 + 1 \cdot 2 = 354$$

5.2.2.5 Interpretation

The final $\phi_K = 354$ reflects the network's integration, combining global (attractor map) and local (rule) sensitivities. Notably:

- Most perturbations (e.g., edge (1,2)) result in Sensitivity$_i$ = 0, indicating that the network's dynamics are robust to many single-edge removals, a sign of high integration where the system remains cohesive.
- Edges (2,4) and (4,2) yield high sensitivity (352), showing that these connections are critical to maintaining the attractor structure. Removing them reduces the number of attractors, collapsing some states into others, which aligns with our expectation for a highly integrated network: perturbations to key edges have a significant impact.
- The small $\Delta K_{\text{rule}} = 2$ adds nuance, capturing subtle changes in the rules generating attractors, enhancing the robustness of our measure.

This demonstration validates our ϕ_K approach, showing how it captures integration through dynamic sensitivity in a computationally efficient manner ($O(E) = 10$ perturbations vs. IIT's $2^{5-1} = 16$ bipartitions). The result ($\phi_K = 354$) contrasts with IIT's $\phi = 0$ for this network, highlighting our method's focus on dynamic sensitivity over irreducible information, making it more suitable for large-scale systems.

5.2.3 Conclusion

Our ϕ_K approach provides a novel, scalable method to measure integration in complex networks, leveraging attractor dynamics and algorithmic complexity. The step-by-step demonstration on a 5-node network illustrates its practical application, capturing integration through sensitivity to perturbations in a way that aligns with the network's intrinsic dynamics. By avoiding IIT's exponential computational cost, ϕ_K paves the way for studying integration in larger, more complex systems, with potential applications in social, economic, and governance contexts.

Chapter 6

Conclusions

Here we have sketched connections and developed first approaches towards a calculus to ϕ metrics based on algorithmic perturbation analysis, which in turn has a solid mathematical foundation that must be further studied. Our computational approach targeted what is referred to as the IIT 3.0, defined as a calculus of probability distributions. Instead of considering distances between statistical distributions, we formulated the problem as a distance in an algorithmic complexity space, properly approximated, in response to perturbations of the system and introduced a meta-test whose answers may provide a guidance on the algorithmic complexity and integrated information of the system. More exploration of the theoretical and practical connections between these theories are still needed.

Interestingly, such a perturbation programmability test–initially inspired by the Turing test (establishing another interesting connection between these n ew theories of consciousness and past ones)– as applied to physical systems, is a working strategy to find explanations for the behaviour of systems. It remains for future work to make conceptual and computational connections to what Oizumi and Tononi et al. called the MIP (Minimum Information Partition) (18) of a system. Having this first version of ϕ_K, we conjecture that MIP definitions also obey and are connected to algorithmic complexity in about the same way, as they should remain based on rules of an algorithmic nature. Thus, the next step is to go further in the application of the test introduced in this paper to discover simple rules that would help to find MIP in a more natural and a faster way. Another possible direction is to systematize the finding of these simple rules and apply more powerful methods to enable computation of larger systems. However, here we have merely established the first principles and the directions that can be explored following these ideas.

Finally, we think that these ideas about self-explanatory systems capable of providing answers to questions about their own behaviour can help in devising techniques to make other methods, in areas such as machine and deep learning, explain their own, often obscure, behaviour.

Appendix A

Schemas of Information

In order to explain the advantages of the generalization of information in the form of schemas computed by simple rules, Table A.1 is introduced. In this Table are shown all possible cases where the pattern {8,9}={1,1} than can be found in the whole output repertoire of the 9-Node system introduced in Figure 4.4 in the main text.

{**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**}
Time: {0.166334} , Memory: {217920}

Table A.1: Reperoire of inputs and outputs where the condition {8,9}={1,1} fullfill for the system shown in Figure 4.4 of the main text

On the right side of the set contained in Table A.1, outputs where {8,9}={1,1} are highlighted in bold. On the left side the 9-length are inputs that yield to outputs containing the desired pattern. On this left side, in bold, are the corresponding inputs that are particularly responsible for causing the desired pattern, that is, all possible patterns for the inputs {1,3,5,6,7}

In order to obtain the results in Table A.1 using the naive (brute force) approach show in Table 4.15 of the main text, it 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 {8,9}={1,1} was carried out. Notice that time and memory used are at least 10 times greater than those used in the ϕ_K approach. These results are shown in the last two rows in Table A.1.

Appendix B

How meta-perturbation test works

Algorithm 1: computeIntegratedInformation

input : AdjacencyMatrix,Dynamic,CurrentState
output: Information Integration Value

```
nodes ← GetNodes(AdjacencyMatrix);
// UPPD: Unrestricted Past Probability Distribution;
// UFPD: Unrestricted Future Probability Distribution;
UPPD ←
 ComputesPastProbabilityDistribution(nodes,CurrentState,∅,Dynamic,am);

UFPD ←
 ComputesFutureProbabilityDistribution(nodes,CurrentState,∅,Dynamic,am);

// am=AdjacencyMatrix; cs=CurrentState;
conceptualSpace ←
 ComputesConceptualSpace(am,Dynamic,cs,UPPD,UFPD);
integratedInformationValue ← 0;
bipartitionsSet ← Bipartitions(conceptualSpace);
foreach bipartition b_i ∈ bipartitionsSet do
    aux ←EMD(b_i, conceptualSpace);
    if aux > integratedInformationValue then
        integratedInformationValue ← aux;
```

Algorithm 2: computeConceptualSpace

input : AdjacencyMatrix,Dynamic,CurrentState,UPPD,UFPD
output: conceptualStructure

```
// UPPD: Unrestricted Past Probability Distribution;
// UFPD: Unrestricted Future Probability Distribution;
nodes ← GetNodes(AdjacencyMatrix);
mechaSet ←Subsets(nodes);
foreach mechanism mecha_i ∈ mechaSet do
    OneConcept
     ←ComputeConceptOfAMechanism(mecha_i, nodes, CurrentState, UPPD, UFPD);

    Append(conceptualSpace,OneConcept)
```

Algorithm 3: computeConceptOfAMechanism

input : mechanism,nodesForPurviews,currentState,pastDistro,futDistro,Dynamic, AdjacencyMatrix
output: concept for current mechanism

```
// nodes where all purviews will be taken from
purviewsSet ←Subsets(nodesForPurviews);

for j ← 1 to Length(PurviewsSet) do
    aPurview ← Part(j, PurviewsSet);
    connected ← FullyConnectedQ(mechanism, APurview);
    if connected then
    // MIP: Maximal Information Partition
    smallAlpha ← ComputesMIP(mechanism, APurview) ;

// APurviewMIP: Purview responsable to cause MIP for current
mechanism;
aPurviewMIP ← smallAlpha("PurviewMIP");
// Following sum is formalized in Figure 4, In Text S2 from
Oizumi(2014);
// cs=CurrentState; am = AdjacencyMatrix;
pastDistribution ←
 ComputesPastProbabilityDistribution(mechanism, cs, aPurviewMIP,
 Dynamic, am);
futureDistribution ←
 ComputesFutureProbabilityDistribution(mechanism, cs, aPurviewMIP,
 Dynamic, am);
ConceptualInfo ←
 EMD(pastDistro, pastDistribution)+EMD(futDistro, futureDistribution);
```

Algorithm 4: ComputesMIP

```
input  : mechanism,purview
output: MIP structure
mechaChildren ← Subsets(mechanism);
PurviewChildren ← Subsets(purview);
ci ← 10000;
ei ← 10000;
foreach mecha m_i ∈ mechaChildren do
    foreach purview p_i ∈ PurviewChildren do
        // cs=CurrentState, am=AdjacencyMatrix;
        pastDistribution ←
         ComputesPastProbabilityDistribution(m_i,cs,p_i,Dynamic,am);
        futureDistribution ←
         ComputesFutureProbabilityDistribution(m_i,cs,p_i,Dynamic,am);
        cei ←
         ComputesCEI(mecha,purview,pastDistribution,futureDistribution);
        if cei ("ci") < ci then
            ci ← cei ("ci");
            pastMIP ← (mecha,purview);
        if cei ("ei") < ei then
            ei ← cei ("ei");
            futMIP ← (mecha,purview);
```

Algorithm 5: computesCEI

input : ChildMecha, ChildPurview, ParentMecha, ParentPurview, ParentPastDistro, ParentFutDistro, UnconstrainedPastDistro, UnconstrainedFutDistro

output: Causal and Effect information values

```
mechaComplement ← ComplementI(ChildMecha, ParentMecha)
purviewComplement ← ComplementI(ChildPurview, ParentPurview)
mechaChildren ← Subsets(mechanism)
PurviewChildren ← Subsets(purview)
foreach mecha m_i ∈ mechaChildren do
    foreach purview p_i ∈ PurviewChildren do
        ChildMecha ← m_i
        ChildPurview ← p_i
        if ChildMecha=∅ then
            pastDistribution ← UnconstrainedPastDistro
            futureDistribution ← UnconstrainedFutDistro
            else if ChildPurview=∅ then
                pastDistribution ← 1
                futureDistribution ← 1
                else if then
                    pastDistribution ←
                     ComputesPastProbabilityDistribution(ChildMecha,ChildPurview,cm,am)
                    futureDistribution ←
                     ComputesFutureProbabilityDistribution(ChildMecha,ChildPurview,cm,am)
        if mechaComplement =∅ then
            pastDistributionComp ← UnconstrainedPastDistro
            futureDistributionComp ← UnconstrainedFutDistro
            else if purviewComplement =∅ then
                pastDistributionComp ← 1
                futureDistributionComp ← 1
                else if then
                    // CPPD=ComputesPastProbabilityDistribution
                    // CFPD=ComputesFutureProbabilityDistribution
                    pastDistributionComp ←
                     CPPD(MechaComplement,PurviewComplement,cm,am)
                    futureDistributionComp ←
                     CFPD(MechaComplement,PurviewComplement,cm,am)
        pastDistribution ← Normalize(pastDistribution *pastDistributionComp)
        futureDistribution ← Normalize(futureDistribution
          *futureDistributionComp)
        ci ← EMD(ParentPastDistro,pastDistribution)
        ei ← EMD(ParentFutDistro,futureDistribution)
        cei ← Min(ci,ei)
```

Algorithm 6: Computes Positions of a Pattern in Outputs

input : Mechanism, Purview, AdjacencyMatrix, CurrentState, Dynamic
output: Positions (indexes) where current state of mechanism is found in the output repertoire

```
// Nodes sending inputs to the mechanism.  Remaining nodes define powers for
pattern distribution.
joinedNames ← Join(Inputs(mechanism))
powers ← Complement(Range(Length(AdjacencyMatrix)), joinedNames) - 1
foreach n_i ∈ powers do
    Append(sumandos, 2^{n_i})
sumandos ← Subsets(sumandos)
foreach s_i ∈ sumandos do
    Append(Aux, Sum(s_i))
sumandos ← Aux
ins ← Inputs(FirstNode(mechanism))
// Compute all possible inputs of defined size resulting in a defined output
(cs).
repertoire ← RepertoireByOutput(Length(ins), Dynamic, cs)
foreach m_i ∈ (mechanism − FirstNode(mechanism)) do
    intersec ← Intersection(ins, Inputs(m_i))
    ins ← ins + (Inputs(m_i) - intersec)
    repertoire ← Combine(repertoire, CreateRepertoire (Inputs(m_i) - intersec))
    repertoire ← FilterRepertoireByOutput(repertoire, cs)
foreach s_i ∈ sumandos do
    foreach r_i ∈ repertoire do
        Append(indexes, Sum(s_i, r_i))
```

Algorithm 7: computesPastProbabilityDistribution

input : Mechanism,Purview,AdjacencyMatrix,CurrentState,Dynamic
output: Probability distribution for a mechanism

```
locations
 ←computesPositionsOfAPattern(Mechanism, CurrentState, Purview, AdjacencyMatrix)

allInputs ← Extract(locations,Purview)
probability ← 1/(Length(Locations))
foreach input in_i ∈ allInputs do
    correctedLocations
     ←FindPatternInInputs(Purview, in_i, Length(AdjacencyMatrix))
probabilityDistribution ←
 ComputesProbabilityForElements(correctedLocations)
```

Algorithm 8: computesFutureProbabilityDistribution

input : Mechanism,Purview,AdjacencyMatrix,CurrentState,Dynamic
output: Probability distribution for a mechanism

locations ←(`FindPatternInInputs(`$Purview, CurrentState,$ `Length(`$AdjacencyMatrix$`))`)-1
allOutputs ← `ComputesOutputs(`locations`)`
allInputs ← `Extract(`locations,$Purview$`)`
probabilityDistribution ← `ComputesProbabilityForElements(`allInputs`)`

Algorithm 9: findPatternInInputs

input : Nodes,WantedPattern,sizeAdjacencyMatrix
output: Finds indexes in input repertoire where nodes fullfill wantedPattern

limit ← $2^{sizeAdjacencyMatrix}$
foreach *node* $n_i \in Nodes$ **do**
  powers ← 2^{n_i-1}
  repetitions ← limit/powers
  longi ← limit/repetitions
  if *if expectedPatt = 1* **then**
    serie ← `CreateSerieEvenNumbers(`repetitions`)`
  else
    serie ← `CreateSerieOddNumbers(`repetitions`)`
  for $i = 1;\ i <$ `Length(`serie`);`$\ i = i + 1$ **do**
    Found ← `Range(((`powers *serie *[i])*-longi)+1,* powers *serie *[i]*`)`

Algorithm 10: computeInputBitProbabilityDistro

input : Nodes,AdjacencyMatrix,Dynamics
output: Bit probability for given nodes

locations ← `FindPatternInInputs(`*Nodes,1,*`Length(`$AdjacencyMatrix$`))`
oneProbability ← `Length(`locations`)` $/2^{\texttt{Length(}AdjacencyMatrix\texttt{)}}$
zeroProbability ← $1 -$ oneProbability

Algorithm 11: computeOutputBitProbabilityDistro

input : Nodes,AdjacencyMatrix,Dynamics
output: Bit probability for given nodes

locations ←
`ComputesPositionsOfAPatternInOutputs(`*Nodes,1,Dynamics,AdjacencyMatrix*`)`

oneProbability ← `Length(`locations`)` $/2^{\texttt{Length(}AdjacencyMatrix\texttt{)}}$
zeroProbability ← $1 -$ oneProbability

Appendix C

Meta-compression

Algorithm 12: Generate Network Data and Export to CSV

Input : maxNodes
Output: CSV file with network data

Initialize networks as an empty list;
if *maxNodes is not provided or is not numeric* **then**
    Set maxNodes to 12;
    Print "maxNodes set to default: 12";
foreach *nodeSize* $\in$ *{5, . . . , maxNodes}* **do**
    Set nodeSize to current value of n;
    Print "Processing nodeSize: " + nodeSize;
    Calculate noEdges as a random integer between $\lfloor \frac{\text{nodeSize}}{2} \rfloor$ and $\lfloor \frac{\text{nodeSize} \times (\text{nodeSize}-1)}{2} \rfloor$;
    Generate a random network with nodeSize nodes and noEdges edges;
    Calculate initial complexity (k0) and length (a0) of binary attractors;
    Initialize sensitivities as an empty list;
    foreach *edge (i, j) in the adjacency matrix* **do**
        if *adjMat[i, j] == 1* **then**
            Set adjMatPert[i, j] to 0;
            Recalculate patterns and binary attractors with the perturbed adjacency matrix;
            if *the perturbed binary attractors are valid* **then**
                Calculate new complexity (k1) and length (a1);
                Calculate deltaK as $|k0 - k1|$, deltaA as $|a0 - a1|$;
                Add deltaK $\times \log_2$(deltaA + 1) to sensitivities;
    Calculate phiK as the mean of sensitivities if sensitivities is not empty, otherwise 0;
    Calculate rules0 as behaviors for each attractor in binAttractors;
    Calculate kRule0 as the mean byte count of rules0;
    Calculate rules1 as behaviors for each attractor in binAttPert with the perturbed adjacency matrix;
    Calculate kRule1 as the mean byte count of rules1;
    Calculate deltaKRule as |kRule0 − kRule1|;
    Calculate phiKRefined as phiK + deltaKRule;
Export networks to a CSV file with columns: "NumberOfNodes", "OutputRepertoire", "AdjacencyMatrix", "OneOutput", "PhiK";

Bibliography

[1] Anderson, P. (1972). More is different. *Science*, 177(4047):393–396. 2

[2] Bailey, D., Borwein, P., and Plouffe, S. (1997). On the rapid computation of various polylogarithmic constants. *Math. Comput.*, 66:903–913. 11

[3] Barabasi, A.-L. and Posfai, M. (2016). *Network Science*. Oxford University Press. 1

[4] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42. 1

[5] Chaitin, G. J. (1966). On the length of programs for computing finite binary sequences. *Journal of the ACM (JACM)*, 13(4):547–569. 3

[6] Chandler, D. G. (1987). *Introduction to Modern Statistical Mechanics*. Oxford University Press. 1

[7] Dayan, P. and Abbott, L. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press. 1

[8] Delahaye, J.-P. and Zenil, H. (2012). Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness. *Applied Mathematics and Computation*, 219(1):63–77. 3

[9] Devine, S. (2006). The application of algorithmic information theory to noisy patterned strings. *Complexity*, 12(2):52–58. 10

[10] Granger, C. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37:424–438. 3

[11] Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press. 32

[12] Iapascurta, V. (2019). Detection of movement towards randomness by applying block decomposition method to a simple model of circulatory system. *Complex Systems*, 28(1). 3

[13] Kitazono, J., Kanai, R., and Oizumi, M. (2018). Efficient algorithms for searching the minimum information partition in integrated information theory. *Entropy*, 20(3):173. arXiv: 1712.06745. 34

[14] Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information and Transmission*, 1(1):1–7. 3

[15] Marabita, F., Deng, Y., Elias, S., Schmidt, A., Ball, G., Tegnér, J., Zenil, H., and Kiani, N. (2019). An algorithmic information calculus for causal discovery and reprogramming systems. *iScience*, S2589-0042(19)30270-6. 3, 4, 8, 11, 20, 28

[16] Mayner, W., Marshall, W., Albantakis, L., Findlay, G., Marchman, R., and Tononi, G. (2017). PyPhi: A toolbox for integrated information theory. arXiv: 1712.09644. 2, 12, 14

[17] Oizumi, M., Albantakis, L., and Tononi, G. (2014a). From the phenomenology to the mechanisms of consciousness: integrated information theory 3.0. *PLoS Comput Biol*, 10(5):e1003588. xi, 2, 5, 11, 12, 13, 14

[18] Oizumi, M., Albantakis, L., and Tononi, G. (2014b). From the phenomenology to the mechanisms of consciousness: integrated information theory 3.0. *PLoS Comput Biol*, 10(5):e1003588. 5, 12, 29, 49

[19] Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK. 4

[20] Rosanova, M., Boly, M., Sarasso, S., Casali, K., Casarotto, S., Bruno, M.-A., Laureys, S., Tononi, G., Casali, A., Gosseries, O., and Massimini, M. (2013). A theoretically based index of consciousness independent of sensory processing and behavior. *Science Translation Medicine*, 5(198):198ra105. 3

[21] Ruffini, G. (2017). An algorithmic information theory of consciousness. *Neuroscience of Consciousness*, 1(nix019). 3

[22] Schreiber, T. (2000). Measuring information transfer. *Physical Review Letters*, 85(2):461–464. 3

[23] Shalizi, C. and Shalizi, K. (2003). Optimal nonlinear prediction of random fields on networks. *Discrete Math Theor Comput Sci*, AB(DMCS):11–30. 10

[24] Shalizi, C. R. and Crutchfield, J. P. (2001). Computational mechanics: Pattern and prediction, structure and simplicity. *Journal of Statistical Physics*, 104(1):817–879. 3, 10

[25] Soler-Toscano, F., Zenil, H., Delahaye, J.-P., and Gauvrit, N. (2014). Calculating kolmogorov complexity from the output frequency distributions of small turing machines. *PLoS ONE*, 9(5):1–18. 3

[26] Solomonoff, R. J. (1964). A formal theory of inductive inference. part i and ii. *Information and Control*, 7(1, 2):1–22, 224–254. 10

[27] Strogatz, S. (2000). *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry, And Engineering (Studies in Nonlinearity)*. CRC Press. 1

[28] Tononi, G. (2004). An information integration theory of consciousness. *BMC neuroscience*, 5:1–22. 5

[29] Tononi, G. (2008). Consciousness as integrated information: a provisional manifesto. *The Biological Bulletin*, 215(3):216–242. 5

[30] Tononi, G., Boly, M., Massimini, M., and Koch, C. (2016). Integrated information theory: from consciousness to its physical substrate. *Nature Reviews Neuroscience*, 17(7):450–461. 5

[31] Ventresca, M. (2018). Using algorithmic complexity to differentiate cognitive states in fmri. In Aiello, L., Cherifi, C., Cherifi, H., Lambiotte, R., Lió, P., and Rocha, L., editors, *Complex Networks and Their Applications VII*, volume 813 of *Studies in Computational Intelligence*. Springer. 3

[32] Villani, C. (2003). *Topics in Optimal Transportation*. American Mathematical Society, Providence, RI. 6

[33] Williams, P. and Beer, R. (2010). Nonnegative decomposition of multivariate information. arXiv:1004.2515. 3

[34] Williams, P. and Beer, R. (2011). Generalized measures of information transfer. arXiv:1102.1507. 3

[35] Zenil, H. (2013a). A behavioural foundation for natural computing and a programmability test. In Dodig-Crnkovic, G. and Giovagnoli, R., editors, *Computing Nature: Turing Centenary Perspective*, volume 7, pages 87–113. Springer SAPERE. 4, 7, 28

[36] Zenil, H. (2013b). A behavioural foundation for natural computing and a programmability test. In *Computing Nature*, pages 87–113. Springer. 7

[37] Zenil, H. (2013c). Testing biological models for non-linear sensitivity with a programmability test. In Nicosia, G., Nolfi, S., Lió, P., Miglino, O., and Pavone, M., editors, *Advances in Artificial Intelligence*, pages 1222–1223. MIT Press. 7, 28

[38] Zenil, H. (2017). Algorithmic data analytics, small data matters and correlation versus causation. In Ott, M., Pietsch, W., and Wernecke, J., editors, *Berechenbarkeit der Welt? Philosophie und Wissenschaft im Zeitalter von Big Data*, pages 453–475. Springer Verlag. 3

[39] Zenil, H., Badillo, L., Hernández-Orozco, S., and Hernández-Quiroz, F. (2018a). Coding-theorem like behaviour and emergence of the universal distribution from resource-bounded algorithmic probability. *International Journal of Parallel Emergent and Distributed Systems*. 3

[40] Zenil, H., Hernández-Orozco, S., Kiani, N., Soler-Toscano, F., Rueda-Toicen, A., and Tegnér, J. (2018b). A decomposition method for global evaluation of shannon entropy and local estimations of algorithmic complexity. *Entropy*, 20(8):605. 3

[41] Zenil, H., Kiani, N., and Tegnér, J. (2016). Methods of information theory and algorithmic complexity for network biology. *Seminars in Cell and Developmental Biology*, 51:32–43. 21

[42] Zenil, H., Kiani, N., and Tegnér, J. (2017a). Low algorithmic complexity entropy-deceiving graphs. *Physical Review E*, 96(012308). 3, 4

[43] Zenil, H., Kiani, N., and Tegnér, J. (2018c). A review of graph and network complexity from an algorithmic information perspective. *Entropy*, 20(8):551. 21

[44] Zenil, H., Kiani, N., and Tegnér, J. (2020). *Algorithmic Information Dynamics: A Computational Approach to Causality in Application to Living Systems*. Cambridge University Press. 3, 4, 8, 11, 20

[45] Zenil, H., Kiani, N., and Tegnér, J. (2019). Algorithmic information dynamics of emergent, persistent, and colliding particles in the game of life. In Adamatzky, A., editor, *From Parallel to Emergent Computing*, pages 367–383. Taylor & Francis / CRC Press. 3

[46] Zenil, H., Schmidt, A., and Tegnér, J. (2017b). Causality, information and biological computation: An algorithmic software approach to life, disease and the immune system. In Davies, P., Walker, S., and Ellis, G., editors, *Information and Causality: From Matter to Life*, pages 244–279. Cambridge University Press. 7, 23, 37, 44

[47] Zenil, H., Soler-Toscano, F., Dingle, K., and Louis, A. (2014). Correlation of automorphism group size and topological properties with program-size complexity evaluations of graphs and complex networks. *Physica A: Statistical Mechanics and its Applications*, 404:341–358. 21