Does a computer think if no one is around to see it?

Ovidiu Cristinel Stoica

Department of Theoretical Physics, National Institute of Physics and Nuclear Engineering – Horia Hulubei, Bucharest, Romania. Email: cristi.stoica@theory.nipne.ro, holotronix@gmail.com

December 18, 2023

Abstract

I show that a computer cannot have unambiguous thoughts, not even about a number. What we believe computers do is our own convention. It may seem objective because we anchor it in the user interface. But many other conventions are possible, and they yield different computations, equally valid according to the principles of Computer Science. I prove that the alternative computations equally happen when a single computation is carried out, and in principle they can be accessed. I exemplify this with a program that computes the result for a given input, and then decodes it into the results for all other possible inputs.

If thinking would be a computation, a computer would have different, possibly opposite thoughts, corresponding to many alternative computations it implements at the same time.

I show probabilistically that the human mind does not have this ambiguity. Therefore, even if the human mind can be simulated by a computer, it cannot be reduced to computation.

1 Introduction

Can an Artificial Intelligence think about something, for example a number, or it's all just our interpretation?

I'm not even asking about the emotions and feelings of the AI, about qualia, all I'm asking is if the AI can think about numbers.

To understand if computers can have unambiguous thoughts, it is important to distinguish what we imagine computers do from what they really do. For this, in Section §2 I will present core principles of Computer Science relevant to this subject, extracted from both *theory* and *practice*.

A key aspect of Computer Science is that a computing system can simulate other computing systems, even if they are performing different computations on different data. We will see that computers implement and run more computations at once, and none of these computations has a special status. Not even the computation shown by the user interface.

In Section §3 I prove the main result, that under generic conditions a computer computes at once the results for all possible inputs.

In Section §4 I exemplify this with a computer program. The program illustrates how a computation for a particular input encodes within itself the computations for all possible inputs. All of the "alternative computations" can be extracted from the result of a single computation. The source code is available for independent verification.

Building on this, in Section §5 I show that, under very general assumptions, the answer to the following question is undefined:

Question 1. *What number is the computer thinking about?*

If we believe that a computer can think of a number, we will see that it would be equally justified to interpret this at the same time as thinking of many other numbers, and there is no objective way by which one of these thoughts is more "real" than the others. Therefore, an AI can't think unambiguously of a number. I illustrate this with several apparently paradoxical and hopefully entertaining thought experiments.

Therefore, if a computer can have a mind, it has many distinct minds unaware of one another, in parallel, simultaneously!

To see whether this also applies to brains, in Section §6 I extend these results to analog and quantum computers, and physical systems in general.

The ambiguity of computation forces us to choose between two options:

Option 1. If a physical system implements a mind, it implements many contradictory minds at the same time.

Option 2. Consciousness and even mere thinking are not computations.

In Section §7 I present a probabilistic argument showing, on an individual basis, that Option 1 fails for the humans, and their minds cannot be reduced to computations.

I conclude with a discussion in Section §8.

2 What do computing systems actually do?

This Section is about the principles of Computer Science. The approach is "experimental", we look at how computation is used in both theory and practice and extract some core principles. In addition, in this Section we fix the mathematical formalism that will be used in the article.

2.1 The core principles of computers

For this article, the most important Thesis of Computer Science is:

Thesis CST (Implementation Independence)**.** Computation is independent of the implementation.

The computation has to be implemented by a physical system, but this can be done in many equally valid ways.

Implementation has two aspects, one of which is often ignored or forgotten, so it is useful to extract them as explicit principles.

First, the material out of which we make the components of a computer doesn't matter. This includes the memory and the logic circuits. All that matters is that computers are able to implement the algorithm, regardless of what materials and mechanisms they use for this. Therefore, we can extract as a core principle of Computer Science the following:

Principle CSP1 (Substrate Independence)**.** Computation is independent of the material substrate of the computing system that implements the computation.

Not only the substrate of the computing system, but also its internal structure and organization are irrelevant to the computation:

Principle CSP2 (Substructure Independence)**.** The physical system implementing a computation is not required to be structured internally in the same way as the computed data.

All that matters is that, once we associate, by any convention, the data to the states of the computing system, the physical changes of the states of the computing system correspond to the algorithm or the computation.

Thesis CST is universally acknowledged in Computer Science, at least in its restrained sense of Principle CSP1. However, it is not always explicitly understood in a sense that includes Principle CSP2.

I think that many people are not fully aware that Principle CSP2 is at the core of Computer Science. But in the rest of this Section, in particular §2.2, I show that Principle CSP2 is necessary and is used everywhere, both in theory and in practice. Without it, there would be no programmable computers.

Note that I don't contradict the importance of the substrate and of the substructure for other domains than Computer Science. All I'm saying is that they are irrelevant for computations and their implementations, according to the very foundations of standard Computer Science. They may be necessary for consciousness, for example as it's manifest in the brain, but this is a different question.

We will see that if one of these core principles would not be true, there would be no Turing equivalence, no Turing universality, no possibility of a programmable computer. Computers would be very specialized, and for every program that you want to use, you'd have to use a strictly dedicated and much more complicated computer.

2.2 Examples

What a computing system computes is a matter of convention. If we change the convention, we change the computation. There is no limit to our choices to interpret the strings of bits. To see this, we will look at various operations that a computer can do, with the eye of an experimentalist who wants to find out the rules governing the computation. The following examples provide "experimental evidence" supporting the role of convention, and the freedom to reinterpret the bits. This freedom is used ubiquitously in the design of the logic circuits, computers, and programs.

Example CSE1. We can use the strings of bits to represent internally numbers, both integer and floating point, in various representations. We can use them to represent text in various encodings, pictures, sounds, movies, in an unlimited number of formats. The data represented as bits don't have to have anything to do with the bits. It is sufficient to be discretizable, and therefore digitizable.

But the role of convention is much greater than this. We may imagine that there is an unambiguous way to assign the values `0` and `1`, or perhaps `F` and `T`, to the two states of each physical bit, but even this isn't true.

Example CSE2. Consider a simple device that has two inputs, let us call them p and q, and an output r. The input p can be a knob with two possible states a and b, the input q can be a switch with two possible states c and d, and the output r can be a pointer with two possible states x and y. Suppose that the output depends on the input as shown in Table 1.

p	q	r
a	c	x
a	d	x
b	c	y
b	d	x

Table 1: Correspondence between the inputs and the output of the device.

We can label each of the inputs and the output with Boolean values like F ("false") and T ("true"), or 0 and 1. For p, we can label a $=$ F and b $=$ T or a $=$ T and b $=$ F, and so on for p and q. So there are eight possible ways to label them. Each of these choices leads to a distinct way to interpret our system as a logic gate that computes a Boolean function $r = f(p, q)$. They are listed in Table 2.

a	b	c	d	x	y	$f(p,q)$
F	T	F	T	F	T	$p \wedge \overline{q}$
F	T	F	T	T	F	$\overline{p} \vee q$
F	T	T	F	F	T	$p \wedge q$
F	T	T	F	T	F	$\overline{p} \vee \overline{q}$
T	F	F	T	F	T	$\overline{p} \wedge \overline{q}$
T	F	F	T	T	F	$p \vee q$
T	F	T	F	F	T	$\overline{p} \wedge q$
T	F	T	F	T	F	$p \vee \overline{q}$

Table 2: The eight possible interpretations of the device as a logic gate.

They are all equally valid interpretations of the same physical gate. □

If we start with any of the interpretations from Table 2, we can find any other one by flipping some of the bits accordingly. This is what I will call a *reinterpretation.*

The internal structure of the gate may suggest a particular interpretation, for example a serial circuit of two push-down buttons and a light bulb may suggest an AND gate. It is tempting to use the physical implementation of an algorithm to claim that it does not implement other algorithms at the same time. And, knowing that in modern computers the logic circuits are built out of transistors, which effectively are microscopic push-down buttons, we may have the impression that only one of the possible interpretations from Table 2 can be "real". However, this would require p to stand for the proposition "the first push-down button is on", q for "the second push-down button is on", and r for "the light bulb is on". But they could as well stand for "the push-down button is off" or "the light bulb is off", so the ambiguity remains.

And indeed, this freedom is used extensively in the design of circuits. Without this, we could not have *universal gates*. For example,

Example CSE3. Any logic circuit can be made exclusively out of `NAND` gates. And yet, a computer made exclusively of `NAND` gates can perform `OR` operations, and any other Boolean operation. The possibility to reinterpret the logic circuits in various ways is what allows the computer to be programmable. Without this, it would be a single-purpose device, and a very large one, since the circuits could not be reused in different ways during the execution of the same program. □

Example CSE4. Consider the *Arithmetic Logic Unit* (ALU) of the computer's processor. The ALU is specialized in performing additions, but it is used for performing subtractions as well, if we interpret the second number as being expressed in the *two's complement representation*. To obtain the two's complement of an integer number represented in binary, we flip each bit and add 1. The ALU can also perform various operations on bits, which require reinterpreting the data differently. □

Note that, while Table 2 contains only *simple interpretations*, consisting of flipping the bits independently, the two's complement representation is an interpretation that is not simply a bit flipping. In fact, there is no limitation of how strings of bits are reinterpreted as representing other strings of bits.

Example CSE5. Error detection and error correction codes are based on encoding the data in a form that can be very different from the original. The encoded data is supposed to carry the same information, but it looks very different from the original data, not only due to the redundancy, but also to various elements added to allow the detection of error. Error correction codes are used for transmitting information and to store it in special memory chips. At the end, to decode the encoded data, it has to be reinterpreted. It is even possible to process data in encoded form. In fact, in quantum computing, the logic circuits themselves have to include error correction (Nielsen and Chuang, 2010). □

Example CSE6. Encryption is another example of reinterpretation of the binary data. The simple `XOR` cipher takes the message represented in binary form and flips some of its bits, specified by a key. This is a *simple reinterpretation*. To decrypt the message one applies the same key, to flip the same bits back to their original form. But the more useful and less breakable encryption methods are much more complicated reinterpretations, that replace bit strings with other bit strings. □

Example CSE7. Moreover, encryption is not limited to static data, but it can be done even for computations. *Fully homomorphic encryption* was considered for three decades the holy grail of cryptography (Marcolla et al., 2022), until the first such encryption scheme was proposed by Gentry in his PhD Thesis (Gentry, 2009). Now we can send the input data to a server, in an encrypted form, together with an *evaluation key*. On the server, the data are processed without having to be decrypted. The evaluation key allows the program on the server to manipulate it blindly. Then the result, still encrypted, is sent back to us, and we can decrypt it. □

Therefore, the same computation can be carried out in multiple forms, and yet all of these forms are the same computation. Moreover, there is no way for someone who looks into the server's circuits to see what data is processed. And yet, the data are processed.

Example CSE8. Reinterpretations of Boolean functions and circuits are ubiquitous in Computer Science (Slepian, 1953; Golomb, 1959). The bits can be flipped and permuted in numerous ways. This is a real thing, and it's applied everywhere from the design of gates and logic circuits and computer architecture to all levels of software engineering. □

Example CSE9. Consider the universal Turing machine (UTM) with two tape symbols and 18 internal states (Rogozhin, 1996). It can simulate any other Turing machine, but its internal structure can only be in 18 possible states. There is another two-symbol 15-states Turing Machine that can be in only 15 different possible internal states, and yet it is universal too (Neary and Woods, 2009). There is also a UTM with only 3 internal states and 9 symbols (Kudlek and Rogozhin, 2002). Also, a very simple cellular automaton named *Rule 110*, with one dimension and two possible states for its cells, is Turing universal (Cook, 2004). Also Conway's *game* *of life* is Turing universal (Gardner, 1970). All universal machines have different internal structures, different combinations of states, and yet they can simulate each other. □

These examples show that indeed not only Principle CSP1, but also Principle CSP2 is necessary. There is no limitation of how to interpret what the states of a computer represent, as long the interpretation leads to a correct result that complies with the convention. And the machine is completely oblivious of the convention, this exists only in our minds.

2.3 Computers

If we ignore the details of implementation, computers can have a simple mathematical description.

A computer with n bits is made of n small subsystems, each of which can be in one of two possible states. If we label the states of the small subsystems by `0` and `1`, the total set of possible states of the computer is

$$\mathcal{S} = \underbrace{\{\mathtt{0},\mathtt{1}\} \times \ldots \times \{\mathtt{0},\mathtt{1}\}}_{n \text{ times}} = \{\mathtt{0},\mathtt{1}\}^n. \tag{1}$$

Then, any state of the computer has the form

$$s = (s_1, \ldots, s_n), \text{ where all } s_j \in \{\mathtt{0},\mathtt{1}\}. \tag{2}$$

The computer transitions from one state into another one in discrete steps. Each such step has a duration T. For simplicity I will choose the unit of time so that $T = 1$, for all computing systems that operate in discrete steps discussed in this article.

If no external cause affects the computer, its transition at any step can be described by a Boolean function

$$\mathcal{F} : \{\mathtt{0},\mathtt{1}\}^n \to \{\mathtt{0},\mathtt{1}\}^n. \tag{3}$$

If at a moment of time the state of the computer is $s \in \mathcal{S}$, after a time interval N the state becomes

$$s_N = \mathcal{F}^N(s) := \underbrace{\mathcal{F}(\ldots\mathcal{F}}_{N \text{ times}}(s)\ldots). \tag{4}$$

Some of the computer's bits can be changed by the user. If the user inputs some data, this changes the computer's state from s to another state

s'. The computer continues to transition according to the same function $\mathcal{F}$, but applied to the modified state, $s' \mapsto \mathcal{F}(s')$.

Since the transition has to depend only on the current state of the computer, the function $\mathcal{F}$ is independent of time. That is, if the computer is functional, whenever it's in a state s, it transitions in the same state $\mathcal{F}(s)$. After N steps, it transitions into the state $\mathcal{F}^N(s)$, provided no external intervention changes its state in the meantime. Usually, the computer contains a digital clock, which is incremented, but even so, $\mathcal{F}$ would not depend on the objective time. It depends only on the clock's state, which in this description is part of the computer's state, so the transition depends only on the current state.

There are states for which $s = \mathcal{F}(s)$. For example, if the computer is turned off, it is in such a state.

It may seem that this description is overly simplified. Computers have sophisticated architectures, with complex logic circuits and different kinds of memory cells, but all of these can be encoded into the function $\mathcal{F}$, which describes how bits change in time.

As seen in §2.2, and in the simplest form in Example CSE2, there is no objective way to assign the values `0` and `1` to the possible states of a bit cell. It is a matter of convention. This fact is not only true, but it is heavily exploited in both practice and theory. Forgetting it may give us the wrong idea about what computers do.

2.4 Computing machines as dynamical systems

By extracting the essential from the description of the computer provided in §2.3, we can obtain a general mathematical description of every computing system.

The basic theoretical model of computation is the Turing machine (Turing, 1937; Davis, 2004). Church and Turing established that any computation that can be performed by the human mind by following an algorithm can also be performed by a Turing machine (Davis, 2004).

A Turing machine consists of a tape and a head. The tape consists of possibly infinitely many identical systems called cells, that can be in various states, $(\ldots, c_1, c_0, c_1, \ldots)$. Each cell state c_j represents a symbol from an alphabet which has a finite number of symbols, including the blank. A cell's state can't change by itself, it can be changed only from the outside, by having its symbol written or deleted by the user or programmer, or by the head of the Turing machine. The head of the Turing machine can be in a finite number of states. Depending on its state, it performs an operation and changes its state. The operations that the head can do are to move to the left or to the right by one cell, to read or modify the symbol in the current cell, or to halt, when it enters into a state representing the end of the computation.

So the total state of the Turing machine includes the state of the tape and that of the head. Let $\mathcal{S}$ be the set of all possible states. The transitions of the Turing machine can be represented, as in the case of the computer, by a function $\mathcal{F}:\mathcal{S}\to\mathcal{S}$. Again, the state can be modified from the outside, when the user inputs data. After that, the machine continues to transition according to the function $\mathcal{F}$, $s\mapsto\mathcal{F}(s)$.

This makes it a discrete dynamical system.

Definition 1. A *discrete dynamical system* $(\mathcal{S},\mathcal{F})$, consists of a *space of states* $\mathcal{S}$ and a *dynamical law* $\mathcal{F}$ that specifies how the system transitions from a state to another one, $\mathcal{F}:\mathcal{S}\to\mathcal{S}$.

Given a state s_0, the *history* originating in s_0 is the ordered set of states

$$h(s_0):=(s_0,\mathcal{F}(s_0),\mathcal{F}^2(s_0),\dots), \tag{5}$$

where each state transitions to the next one.

A *morphism* between two discrete dynamical systems $(\mathcal{S},\mathcal{F})$ and $(\mathcal{S}',\mathcal{F}')$ is a function $\alpha:\mathcal{S}\to\mathcal{S}'$, so that for any state $s\in\mathcal{S}$,

$$\mathcal{F}'(\alpha(s))=\alpha(\mathcal{F}(s)). \tag{6}$$

If the function α from equation (6) is invertible, its inverse α^{-1} defines a morphism from $(\mathcal{S}',\mathcal{F}')$ to $(\mathcal{S},\mathcal{F})$. In this case α is called *isomorphism*. An *automorphism* is an isomorphism between a dynamical system and itself.

A *restriction* of a discrete dynamical system $(\mathcal{S},\mathcal{F})$ is a dynamical system $(\widetilde{\mathcal{S}},\mathcal{F})$ obtained by restricting the set of states to a subset $\widetilde{\mathcal{S}}\subseteq\mathcal{S}$. This includes the situation when a state from $\widetilde{\mathcal{S}}$ may transition to another state in $\widetilde{\mathcal{S}}$ indirectly, via intermediate states that are not in $\widetilde{\mathcal{S}}$. In this case the dynamical law of $\widetilde{\mathcal{S}}$ is not simply $\mathcal{F}|_{\widetilde{\mathcal{S}}}$, because it includes indirect transitions of the form $\mathcal{F}^n|_{\widetilde{\mathcal{S}}}$, with n the number of intermediary transitions.

A *partial morphism* between two discrete dynamical systems $(\mathcal{S},\mathcal{F})$ and $(\mathcal{S}',\mathcal{F}')$ is a morphism between a restriction of $(\mathcal{S},\mathcal{F})$ and $(\mathcal{S}',\mathcal{F}')$.

A Turing machine $\mathcal{M}$ can simulate another Turing machine $\mathcal{M}'$ if each state of $\mathcal{M}'$ can be associated to one or more states of $\mathcal{M}$, so that whenever

$\mathcal{M}$ is in a state that corresponds to a state of $\mathcal{M}'$, it transitions (in one or more steps so that no intermediate step is associated to a state of $\mathcal{M}'$) into the state that corresponds to the state in which $\mathcal{M}'$ would transition.

Definition 2. A Turing machine *simulates* another Turing machine if there is a partial morphism between the first Turing machine and the second one. Two Turing machines that can simulate one another are said to be *Turing equivalent*. A Turing machine that can simulate any other Turing machine is called a *universal Turing machine*.

The machine that implements the simulation doesn't necessarily transition directly between two states that represent states of the simulated machine, it may do it via intermediate states that don't represent states of the simulated machine. This possibility is rather the rule, and it's captured in the notions of restriction and partial morphism from Definition 1.

Thesis CST can be restated in a way that makes explicit its main aspect:

Thesis CST′ (Equal authenticity of simulations)**.** A computation done by a computing system is as authentic as the same computation done by any of the simulations of that computing system.

Or, put it differently,

> *A computation done by a computing system is as* conventional *as the same computation done by any of the simulations of that computing system.*

Definition 2 of simulation involves only the states and transitions, and ignores the details of each state or of how the Turing machine works. But these details may be very different. The number of possible states of the cell, and the number of possible internal states of the head, as well as the way it works, may be very different, as seen in Example CSE9.

Question 2. *Shouldn't we use a more detailed description of the states, and not, as in Definition 1, just how they transition into other states?*

Answer. This may be useful in various situations. But imposing such a restriction in the definition of simulation or computation or implementation would be incompatible with Turing universality. A universal Turing machine can simulate any other Turing machine. Therefore, a simulation is just a partial morphism of dynamical systems as in Definition 1, regardless of the internal structure.

If the partial morphism would not be about states and transitions only, but it would also be limited by the structures of the states, then the possible simulations would be limited. No Turing machine would be able to simulate another Turing machine that has more internal states or more tape symbols. This is not the way simulation of Turing machines happens in Computer Science. □

A computer can be seen as a Turing machine with finite length tape, which consists of its memory bits. The operations done by the head correspond to the fetch-decode-execute cycle of the processor, and the internal states are represented by register bits. But because all that matters are the transitions, in the following we will not need to make these distinctions, the description from §2.3 is sufficient.

The description of computing systems as dynamical systems is not very usual, but it is correct and it was used for a long time, for example in (Gandy, 1980; Toffoli, 1980; Fredkin and Toffoli, 1982). Systematic developments of descriptions of computing systems as dynamical systems can be found in (Sieg, 2002; Giunti, 1997).

The formulation given here in terms of dynamical systems may have inessential differences from other presentations, but it is as simple as possible, and it is suited to the proofs that will come later in the article.

2.5 Implementation as convention

An *algorithm* is a list of instructions to be carried out step-by-step on some data. The algorithm can be described as a discrete dynamical system: each stage in which the data can be during the computation corresponds to a state, and each step in the computation corresponds to a transition. An algorithm specifies how the computations should be carried out on more possible values of the data. Computations are particular histories of the dynamical system, as in Definition 1.

A computation needs a machine to implement it. The machine doesn't "know" about the computation. We assign data to the states of the machine by convention, and we make sure that its transitions correspond to valid steps in the computation. In other words, we establish a morphism between the machine and the algorithm, seen as dynamical systems. The convention exists in our minds, not in the computers.

Definition 3. An *interpretation* of a physical state as data is an assignment of data to the physical state. An interpretation of a physical dynamical system as a computing system for an algorithm consists of interpreting its physical states as data processed by the algorithm, so that the transitions of the physical system correspond to steps made according to the algorithm.

An *implementation* is a partial morphism of dynamical systems between a physical system, called *computing system*, and an algorithm (Figure 1).

$$\begin{tikzcd}
{} \arrow[r, dashed] & \text{data stage } j \arrow[r, "\text{step } j \text{ of the}", "\text{computation}"'] & \text{data stage } j+1 \arrow[r, dashed] & {} \\
{} \arrow[r, dashed] & \text{physical state } j \arrow[u, "\text{interp.}"] \arrow[r, "\text{physical}", "\text{transition}"'] & \text{physical state } j+1 \arrow[u, "\text{interp.}"] \arrow[r, dashed] & {}
\end{tikzcd}$$

Figure 1: Implementation of an algorithm by a physical system.

Lemma 1 (Many Reinterpretations)**.** Any computing system implements all algorithms implemented by all computing systems it simulates.

Proof. Let the computing system $(\mathcal{S},\mathcal{F})$ implement an algorithm $(\mathcal{S}_A,\mathcal{F}_A)$, $\alpha:\mathcal{S}\to\mathcal{S}_A$. Let $\beta:\widetilde{\mathcal{S}}\to\mathcal{S}'$, where $\widetilde{\mathcal{S}}\subseteq\mathcal{S}$, be a simulation by $(\mathcal{S},\mathcal{F})$ of another computing system $(\mathcal{S}',\mathcal{F}')$ that implements an algorithm $(\mathcal{S}'_A,\mathcal{F}'_A)$, $\alpha':\widetilde{\mathcal{S}}'\to\mathcal{S}'_A$, where $\widetilde{\mathcal{S}}'\subseteq\beta(\widetilde{\mathcal{S}})$.

$$\begin{tikzcd}
(\mathcal{S}_A,\mathcal{F}_A) & & (\mathcal{S}'_A,\mathcal{F}'_A) \\
(\mathcal{S},\mathcal{F}) \arrow[u, "\alpha", "\text{implem.}"'] \arrow[urr, dashed, "\alpha'\circ\beta", "\text{implementation}"'] \arrow[rr, "\beta", "\text{simulation}"'] & & (\mathcal{S}',\mathcal{F}') \arrow[u, "\alpha'", "\text{implem.}"']
\end{tikzcd}$$

Figure 2: Any computing system implements all algorithms implemented by all computing systems it simulates.

Then, since these partial morphisms compose, $\alpha'\circ\beta:\widetilde{\mathcal{S}}\to\mathcal{S}'_A$ is an implementation of the algorithm $(\mathcal{S}'_A,\mathcal{F}'_A)$ by the computing system $(\mathcal{S},\mathcal{F})$ (Figure 2). □

Similarly, if $\alpha:\mathcal{S}\to\mathcal{S}_A$ is an implementation by the computing system $(\mathcal{S},\mathcal{F})$ of an algorithm $(\mathcal{S}_A,\mathcal{F}_A)$, and $\rho:\mathcal{S}_A\to\mathcal{S}'_A$ is a reinterpretation of $(\mathcal{S}_A,\mathcal{F}_A)$ as another algorithm $(\mathcal{S}'_A,\mathcal{F}'_A)$, $\rho\circ\alpha$ is another implementation (Figure 3).

Indeed, in §2.2 we have seen several examples of computing systems implementing more computations at once.

$$\begin{tikzcd}
(\mathcal{S}_A, \mathcal{F}_A) \arrow[r, "\rho", "\text{reinterpretation}"'] & (\mathcal{S}'_A, \mathcal{F}'_A) \\
(\mathcal{S}, \mathcal{F}) \arrow[u, "\alpha", "\text{implem.}"'] \arrow[ur, dashed, "\rho\circ\alpha", "\text{implementation}"'] &
\end{tikzcd}$$

Figure 3: Any computing system implements all reinterpretations of an algorithm that it implements.

2.6 But still, shouldn't computations depend on structure?

One may raise the following objection:

Objection 1. Principle CSP2 claims that the internal structure of the state of the machine is irrelevant. But not everybody agrees. For example, in (Chalmers, 1994, p. 393-394), Chalmers states

> The states in most computational formalisms have a combinatorial structure: a cell pattern in a cellular automaton, a combination of tape-state and head-state in a Turing machine, variables and registers in a Pascal program, and so on.

So he proposes *combinatorial-state automata*, which are dynamical systems whose states may have more components, and can be represented as vectors (Chalmers, 1994, 1996, 2011). The physical system that implements the computation consists of parts, the data processed by a computation also consist of parts, and the implementation should not only associate data to physical states, but also parts of the data to parts of the physical state.

In Chalmers's words (Chalmers, 2011 p. 333)

> the relation between an implemented computation and an implementing system is one of isomorphism between the formal structure of the former and the causal structure of the latter.

Reply 1. There is some truth in this. When we build a simulation of a computing system by another one, we guide ourselves by patterns and combinatorial structures. Then we map a structure we find in the simulated machine to one in the machine that simulates is. But we don't need to use this more than as a crutch for our own limitations. And this doesn't prevent Principle CSP2 from being true in Computer Science

If, for example, we want to simulate a computer by using a cellular automaton (CA), we will look for patterns. A certain stable pattern may represent a `0`, and another one may represent a `1`. Some interactions between patterns will then represent logic gates and so on. But there is no rule of how to choose these patterns. There is no limitation, as long as a morphism between some of the states of the CA and the computer's states is found. Suppose we simulate the same CA on a computer. This time we will represent each cell of the automaton as a bit, or as more bits, depending on the number of possible states in which each cell can be. But in the CA, we represented a bit by a pattern made of more cells. So if we simulate on the CA a computer that simulates the CA itself, we will have to simulate each cell of the CA as a pattern made of more cells. What if we simulate on the CA a computer that simulates the CA that simulates the computer that simulates the CA that simulates the computer that simulates the CA? How can we then say that the simulation and the implementation have the same structure? And how is this relevant for the data encoded by such structures?

Moreover, there is no real limitation to what we may call patterns or combinatorial structure, to the extent that the idea is superfluous. Consider any simulation that ignores the combinatorial structure. Then take the combinatorial structure of the simulated, and go back with the partial morphism to find out the states from the system that simulates it. Whatever we will find, it can be seen as a pattern or a combinatorial structure. We can decode, with more or less effort, the simulated combinatorial structure as a pattern in the system that simulates it. This would make irrelevant any restriction that the combinatorial structure may add.

More important, the requirement that the morphisms realizing the implementations preserves the combinatorial structures imposes extreme constraints on Computer Science, and implicitly on all current research programs of Artificial Intelligence. I will show this in Proposition 1. □

Proposition 1. If a computation could be implemented only by physical systems whose states are structured in the same way as the computed data, there would be no universal computing systems.

Proof. This should already be clear from the Examples from Section §2.2 and Reply 1. For example, for a Turing machine to simulate another Turing machine with more symbols or internal states, it would have to reinterpret its combinatorial structure as the possibly completely different combinatorial structure of the simulated Turing machine. Therefore, if the partial morphism isn't blind to the combinatorial structure, there is no simulation. □

A universal Turing machine can simulate all other Turing machines, regardless of the number of symbols or of internal head-states they have. And it can implement any computation. This can work only if a computing system can simulate another one as in Definition 2, regardless of their internal structure or organization. If the internal structure would matter, then there would be no Turing universality. Such a limitation would make the entire edifice come crumbling down. Most importantly, the elimination of universality would make it difficult or impossible to claim that we can create an artificial human-like mind on a computer. Moreover, since humans can follow any algorithm step-by-step, their minds are clearly not subject to such a limitation.

I'm not saying that the structure of the states shouldn't matter, all I'm saying it doesn't matter in Computer Science as it is. Therefore, a model that takes structure into account is not quite a computational model, but something else, based on "structure-dependent computations". Something that is allowed to violate Principle CSP2. I will return to this question in the articles (Stoica, 2023a,b).

3 Theorem: Everything at once

Now I will prove that there are situations of maximal ambiguity of what computation a computing system implements, and these situations are not exceptional.

Consider a program that accepts as input any combination of n bits, and computes a result. We assume that the program stores the input in a set of n bits of its memory, and writes the output in a different set of m bits of its memory. In addition, the memory of the program contains a number of c bits, hereby called *clock*, changing in time so that it doesn't return to a previous state. Suppose furthermore that during and after the computation the result is kept in the memory. Therefore, after a sufficiently large number of steps N chosen so that the computation is finished regardless of the input, the result can be found in its memory location. Then, the function $\mathcal{F}$ from equation (3) has the form

$$\mathcal{F}(\underbrace{b_1,\ldots,b_n}_{\text{input data}},\underbrace{b_{n+1},\ldots,b_{n+m}}_{\text{output data}},\underbrace{b_{n+m+1},\ldots,b_{n+m+c}}_{\text{clock}},\underbrace{b_{n+m+c+1},\ldots}_{\text{rest of memory}}). \tag{7}$$

These constraints are not very severe, in fact usual computations satisfy them. But being aware of them will make the proof easier and reveal a maximal ambiguity.

Theorem 1. *The final state of the computation for an input can be reinterpreted as the final state of the computation for any other input of the same algorithm.*

Proof. A transition corresponding to a step of a computation has the form

$$\begin{aligned} &\mathcal{F}(\underbrace{b_1,\dots,b_n}_{\text{input data}},\underbrace{b_{n+1},\dots,b_{n+m}}_{\text{output data}},\ \underbrace{b_{n+m+1},\dots,b_{n+m+c}}_{\text{clock}},\underbrace{b_{n+m+c+1},\dots}_{\text{rest of memory}}) \\ =&(\underbrace{b_1,\dots,b_n}_{\text{input data}},\underbrace{b'_{n+1},\dots,b'_{n+m}}_{\text{output data}},\ \underbrace{b'_{n+m+1},\dots,b'_{n+m+c}}_{\text{clock}},\underbrace{b'_{n+m+c+1},\dots}_{\text{rest of memory}}). \end{aligned} \tag{8}$$

Let us collect the transitions corresponding to all 2^n possible inputs in a set of histories. Even if not all inputs are accepted by the algorithm, the program will give an error message and halt, but the clock will keep changing.

The 2^n possible initial states form a set of states $\mathcal{S}_0$. Let $\mathcal{S}_j := \mathcal{F}^j(\mathcal{S}_0)$ be the possible states resulting after j steps from a state from $\mathcal{S}_0$. Since the bits from $n+m+1$ to $n+m+c$ are the binary representation of j,

$$\mathcal{S}_j \cup \mathcal{S}_{j'} = \varnothing \text{ for } j \neq j'. \tag{9}$$

Also, since the first n bits remain unchanged, the functions

$$\mathcal{F}|_{\mathcal{S}_j} : \mathcal{S}_j \to \mathcal{S}_{j+1} \tag{10}$$

are bijective for all values of j. Therefore, since $\mathcal{S}_0$ has 2^n elements, each set $\mathcal{S}_j$ has exactly 2^n elements. Since all $\mathcal{S}_j$ are disjoint, the histories do not return to a previous state. Since the functions $\mathcal{F}|_{\mathcal{S}_j}$ are bijective, the histories do not intersect one another, as shown in Figure 4.

$$\begin{tikzcd}
\mathcal{S}_0 & \mathcal{S}_1 & \cdots & \mathcal{S}_j & \mathcal{S}_{j+1} & \cdots & \mathcal{S}_N \\
s_1 \arrow[r] & \mathcal{F}(s_1) \arrow[rr, dotted] & & \mathcal{F}^j(s_1) \arrow[r] & \mathcal{F}^{j+1}(s_1) \arrow[rr, dotted] & & \mathcal{F}^N(s_1) \\
s_2 \arrow[r] & \mathcal{F}(s_2) \arrow[rr, dotted] & & \mathcal{F}^j(s_2) \arrow[r] & \mathcal{F}^{j+1}(s_2) \arrow[rr, dotted] & & \mathcal{F}^N(s_2) \\
\vdots & \vdots & & \vdots & \vdots & & \vdots \\
s_{2^n} \arrow[r] & \mathcal{F}(s_{2^n}) \arrow[rr, dotted] & & \mathcal{F}^j(s_{2^n}) \arrow[r] & \mathcal{F}^{j+1}(s_{2^n}) \arrow[rr, dotted] & & \mathcal{F}^N(s_{2^n})
\end{tikzcd}$$

Figure 4: Parallel histories corresponding to the possible computations.

Then, the permutations of the set $\mathcal{S}_0$ are in one-to-one correspondence with the permutations of the set of histories that preserve the clock. These are in fact automorphisms of the computing system. It follows that, for any two possible inputs of the program, there are at least $(2^n - 2)!$ reinterpretations that reinterpret the result of the computation for the first input as the result of the computation for the second input.

Moreover, the resulting transformation on the set $\mathcal{S}_0 \cup \ldots \cup \mathcal{S}_N$ preserves the form of $\mathcal{F}$. Let $s_k \neq s_{k'} \in \mathcal{S}_0$ be the two initial states to be swapped. Let $\mathcal{R}_0 : \mathcal{S}_0 \to \mathcal{S}_0$ be a permutation of $\mathcal{S}_0$ so that $\mathcal{R}_0(s_k) = s_{k'}$ and $\mathcal{R}_0(s_{k'}) = s_k$. For any $j \in \{0, \ldots, N\}$ define the function $\mathcal{R}_j : \mathcal{S}_j \to \mathcal{S}_j$, $\mathcal{R}_j(\mathcal{F}^j(s_l)) := \mathcal{F}^j(\mathcal{R}_0(s_l))$ for all $s_l \in \mathcal{S}_0$. Then, $\mathcal{R}_j$ is a permutation of $\mathcal{S}_j$ that swaps the states corresponding to the two input states, $\mathcal{R}_j(\mathcal{F}^j(s_k)) = \mathcal{F}^j(s_{k'})$ and $\mathcal{R}_j(\mathcal{F}^j(s_{k'})) = \mathcal{F}^j(s_k)$. Since the sets $\mathcal{S}_j$ and $\mathcal{S}_{j'}$ are disjoint for $j \neq j'$, we can define the function $\mathcal{R} : \mathcal{S}_0 \cup \ldots \cup \mathcal{S}_N \to \mathcal{S}_0 \cup \ldots \cup \mathcal{S}_N$ so that its restriction to every $\mathcal{S}_j$ is $\mathcal{R}_j$, which satisfies $\mathcal{R}(\mathcal{F}(s)) = \mathcal{F}(\mathcal{R}(s))$. Therefore, the reinterpretation reinterprets the result of the computation for the input bits of s_k as the result of the computation for the input bits of $\mathcal{R}(s_k)$. This shows that the reinterpretation is an automorphism, so the algorithm is the same, only the inputs and the resulting computations are permuted. □

One may want to raise some objections, like the following.

Objection 2. The conditions to be satisfied by the kind of program from Theorem 1 are too restrictive. No real world program satisfies them.

Reply 2. Actually, all modern computers have a clock, and since the function $\mathcal{F}$ describes the evolution of the totality of the computer, the clock should be included in its data. In reality the clock bits are split between the system clock and a separate register incremented at any oscillator cycle, that generates an interrupt making the CPU change the system time. But together these components change in time without returning to a previous state. About keeping the input data in the memory, almost any program does this by default. If not, this minor change that can be made easily. Therefore, all conditions are usually satisfied. □

The readers can verify Theorem 1 by themselves by performing the concrete experiment described in Section §4.

Observation 1. The user has the impression that the computer executes a definite computation, unambiguously. After all, the computer is one of the most precise tools. This impression is due to the following facts:

1. The user interface establishes a particular convention between the user and the computer.
2. The logic gates and circuits are designed to take into account this convention for the inputs and the outputs.

As seen in §2.2, the freedom to change the convention between input and output is unlimited, as long as it returns to the initial convention. □

These results can be seen as a variant of the *triviality argument*. A concrete and verifiable variant, derived explicitly from the theoretical and practical principles of Computer Science. I discuss this in Appendix §A.

4 Experimental test: The Partition Problem

I will now demonstrate experimentally that the alternative computations really happen. The reader can reproduce the experiment independently. I will choose a particular program, but this is for exemplification only, similar experiments can be done for many other algorithms.

Imagine two friends who want to divide an amount of money consisting of three 5¢ coins, two 10¢ coins, and a 25¢ coin. To divide them fairly, each one of the two friends should receive the same total amount of money, as in Figure 5.

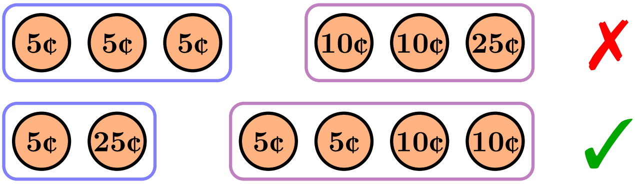


Figure 5: The partition problem: unfair and fair partitions of coins.

This is the *partition problem*. More generally, given a list of positive integer numbers $(x_1, \ldots, x_n)$, can it be partitioned into two lists, so that the total of the numbers in the first list is equal to the total of the numbers in the second list?

To solve the partition problem, one can make a program that tries all 2^n possible partitions into two lists, although there are methods to solve the problem without verifying all of the cases (Hayes, 2002).

But I will do something different:

I will make a program that verifies only one of these cases, and obtains the results of all other cases by reinterpreting the data bits.

To show off, I will make the program verify the partition of $(x_1, \ldots, x_n)$ into the empty list $()$ and the list $(x_1, \ldots, x_n)$. The sums of the two lists will be respectively 0 and $\sum_{j=1}^{n} x_j > 0$, so they are definitely not equal. But if I am right that all other possible cases are computed in parallel, we should be able to make a program that verifies one of them, and then finds the result of all the other cases by changing the interpretation. This includes finding the fair partitions, if they exist!

Let us arrange the numbers from the list in a vector of dimension n,

$$\mathsf{v} = (x_1, \ldots, x_n)^{\intercal} \in \mathbb{Z}^n, \tag{11}$$

where the *transposition* operator $\intercal$ interchanges the rows and the columns by flipping the matrix around the main diagonal. A partition in two can be specified by another list of n numbers, all being either $+1$ or -1,

$$\omega = (w_1, \ldots, w_n)^{\intercal} \in \{+1, -1\}^n. \tag{12}$$

To solve the problem, the program can verify if there is a list ω so that

$$\sum_{j=0}^{n} w_j x_j = 0. \tag{13}$$

Let us construct a matrix from the list of numbers $(x_1, \ldots, x_n)$,

$$\mathbf{U} = \begin{pmatrix} 1 & x_1 & x_2 & \ldots & x_n \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix} = \begin{pmatrix} 1 & \mathsf{v}^{\intercal} \\ \mathsf{0}_{n,1} & \mathbf{I}_n \end{pmatrix}, \tag{14}$$

where in the right-hand side we use block matrices. $\mathsf{0}_{n,1}$ is the vector with all n components zero, and $\mathbf{I}_n$ is the $n \times n$ identity matrix.

We can obtain the sum $\sum_{j=0}^{n} w_j x_j$ by a matrix multiplication,

$$\begin{pmatrix} 1 & \mathsf{v}^{\intercal} \\ \mathsf{0}_{n,1} & \mathbf{I}_n \end{pmatrix} \begin{pmatrix} 0 \\ \omega \end{pmatrix} = \begin{pmatrix} \mathsf{v}^{\intercal}\omega \\ \omega \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} w_j x_j \\ \omega \end{pmatrix}. \tag{15}$$

Therefore, our verification finds that ω partitions the list of numbers $(x_1, \ldots, x_n)$ into two lists having the same sum if and only if the first element of the vector from the right-hand side of eq. (15) is 0.

Of course, it would be much easier to just compute the sum $\sum_{j=0}^{n} w_j x_j$ directly and compare it with 0. But I don't just want to verify this, I want to do it in a way that can be used for proving the existence of the alternative computations in a clear way. The matrix approach explained above will allow this.

Then, the program basically implements a matrix multiplication.

The program will verify only if a particular partition, say given by $\omega_0 = (w_{01}, \dots, w_{0n})^\intercal \in \{+1, -1\}^n$, solves the problem. But how can we access the corresponding results for the other cases?

For any other vector $\omega \in \{+1, -1\}^n$, we are looking for a transformation $\mathbf{R}_\omega$ that allows us to reinterpret or to decrypt the result of the computation applied to ω from the result of the current computation for ω_0,

$$\begin{pmatrix} \sum_{j=1}^{n} w_j x_j \\ \omega \end{pmatrix} = \mathbf{R}_\omega \begin{pmatrix} \mathsf{v}^\intercal \omega_0 \\ \omega_0 \end{pmatrix} = \mathbf{R}_\omega \begin{pmatrix} \sum_{j=1}^{n} w_{0j} x_j \\ \omega_0 \end{pmatrix}. \tag{16}$$

Notice that any vector $\omega = (w_1, \dots, w_n)^\intercal \in \{+1, -1\}^n$ can be obtained from ω_0 by multiplying it with a matrix $\mathbf{S}_\omega$,

$$\omega = \mathbf{S}_\omega \omega_0, \tag{17}$$

where

$$\mathbf{S}_\omega = \begin{pmatrix} w_1 w_{01} & 0 & \dots & 0 \\ 0 & w_2 w_{02} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & w_n w_{0n} \end{pmatrix}. \tag{18}$$

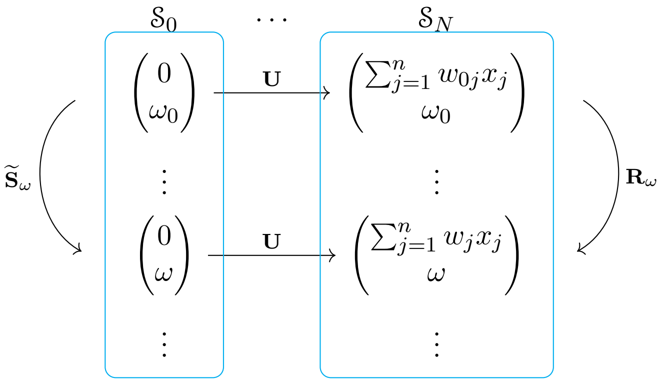
We also define the transformation

$$\widetilde{\mathbf{S}}_\omega := \begin{pmatrix} 1 & \mathsf{0}_{1,n} \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}. \tag{19}$$

With the notation from the proof of Theorem 1, the transformation $\widetilde{\mathbf{S}}_\omega$ from (19) is applied to the set $\mathcal{S}_0$, and permutes its elements. It also permutes the histories, and the permutation of the histories induces, at the end of the computation, the transformation $\mathbf{R}_\omega$, as in Figure 6. For the result, we need to find the transformation $\mathbf{R}_\omega$ of the set $\mathcal{S}_N$, where N is the internal clock time at the end of the computation.

Then, from Figure 6 we see that

$$\mathbf{R}_\omega = \mathbf{U} \widetilde{\mathbf{S}}_\omega \mathbf{U}^{-1}, \tag{20}$$



$\mathcal{S}_0$
$\cdots$
$\mathcal{S}_N$
$\begin{pmatrix}0\\ \omega_0\end{pmatrix}$
$\mathbf{U}$
$\begin{pmatrix}\sum_{j=1}^n w_{0j}x_j\\ \omega_0\end{pmatrix}$
$\widetilde{\mathbf{S}}_\omega$
$\mathbf{R}_\omega$
$\begin{pmatrix}0\\ \omega\end{pmatrix}$
$\mathbf{U}$
$\begin{pmatrix}\sum_{j=1}^n w_{j}x_j\\ \omega\end{pmatrix}$

Figure 6: Parallel histories corresponding to the possible computations.

where $\mathbf{U}^{-1}$ is the *inverse* of the matrix $\mathbf{U}$,

$$\mathbf{U}\mathbf{U}^{-1} = \mathbf{U}^{-1}\mathbf{U} = \mathbf{I}_{n+1}. \tag{21}$$

We can compute the result from equation (20) explicitly. From equation (14), the inverse of the matrix $\mathbf{U}$ is

$$\mathbf{U}^{-1} = \begin{pmatrix} 1 & -\mathsf{v}^\intercal \\ \mathsf{0}_{n,1} & \mathbf{I}_n \end{pmatrix}. \tag{22}$$

Since

$$\begin{aligned}
\begin{pmatrix}\mathsf{v}^\intercal\omega\\ \omega\end{pmatrix} &\overset{(15)}{=} \mathbf{U}\begin{pmatrix}0\\ \omega\end{pmatrix} \overset{(17)}{=} \mathbf{U}\begin{pmatrix} 1 & \mathsf{0}_{1,n} \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\begin{pmatrix}0\\ \omega_0\end{pmatrix} \\
&\overset{(21)}{=} \mathbf{U}\begin{pmatrix} 1 & \mathsf{0}_{1,n} \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\mathbf{U}^{-1}\mathbf{U}\begin{pmatrix}0\\ \omega_0\end{pmatrix} \\
&\overset{(15)}{=} \mathbf{U}\begin{pmatrix} 1 & \mathsf{0}_{1,n} \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\mathbf{U}^{-1}\begin{pmatrix}\mathsf{v}^\intercal\omega_0\\ \omega_0\end{pmatrix} \\
&\overset{(22)}{=} \begin{pmatrix} 1 & \mathsf{v}^\intercal\mathbf{S}_\omega \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\begin{pmatrix} 1 & -\mathsf{v}^\intercal \\ \mathsf{0}_{n,1} & \mathbf{I}_n \end{pmatrix}\begin{pmatrix}\mathsf{v}^\intercal\omega_0\\ \omega_0\end{pmatrix} \\
&= \begin{pmatrix} 1 & \mathsf{v}^\intercal\mathbf{S}_\omega - \mathsf{v}^\intercal \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\begin{pmatrix}\mathsf{v}^\intercal\omega_0\\ \omega_0\end{pmatrix},
\end{aligned} \tag{23}$$

we obtain that the transformation $\mathbf{R}_\omega$ from (16) can be taken as

$$\mathbf{R}_\omega = \mathbf{U}\begin{pmatrix} 1 & \mathsf{0}_{1,n} \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}\mathbf{U}^{-1} = \begin{pmatrix} 1 & \mathsf{v}^\intercal\mathbf{S}_\omega - \mathsf{v}^\intercal \\ \mathsf{0}_{n,1} & \mathbf{S}_\omega \end{pmatrix}. \tag{24}$$

When applied as in equation (16), the transformation $\mathbf{R}_\omega$ gives a vector having on the first position the number $\sum_{j=1}^{n} w_j x_j$, which is zero if and only if the partition ω solves the partition problem.

Therefore, to get the result of the computation for another partition ω, we can reinterpret the result of the computation for the original partition ω_0 by using the transformation $\mathbf{R}_\omega$.

We can extend the program to check by direct computation the results obtained by using $\mathbf{R}_\omega$ for all 2^n possible values of ω.

As promised, it is sufficient that the program verifies the partition

$$\omega_0 = (+1, \ldots, +1)^\intercal, \tag{25}$$

which clearly isn't even a solution. And yet, by using equation (16), it is still possible to obtain from this the correct answer for any other partition, including those that solve the problem.

Remark 1. This confirms what we already know from Theorem 1: that when a program computes the result for some input, it also computes the result for other inputs, and, under the proper setup, it can do it for all possible inputs. To actually get the result of the alternative computations, we had to actively apply the transformation that corresponds to the reinterpretation, as we did here. □

All matrices involved, including $\mathbf{R}_\omega$, $\mathbf{U}$, and even $\mathbf{U}^{-1}$, and all vectors involved, including v, ω_0, and ω, have only integer elements. The operations involved in the algorithm always produce vectors and matrices with integer elements. Therefore, there is no approximation, hence no loss of information. We store an integer in binary form, on a fixed number of bits m. The vector $\left(\begin{smallmatrix}0\\ \omega_0\end{smallmatrix}\right)$ is expressed on $(n+1)m$ bits, even though all its information is contained in n bits, because the program transforms it into $\left(\begin{smallmatrix}\sum_{j=1}^{n} w_{0j} x_j\\ \omega_0\end{smallmatrix}\right)$, as in equation (15). This is a Boolean operation on $(n+1)m$ bits, and it's reversible, because the matrix $\mathbf{U}$ is invertible.

To find the result of an alternative computation, we apply the transformation $\mathbf{R}_\omega$ to the result, to obtain $\left(\begin{smallmatrix}\sum_{j=1}^{n} w_j x_j\\ \omega\end{smallmatrix}\right)$, as in equation (16). At the bit level, this Boolean function is reversible too.

Here we should distinguish two kinds of transformations. The matrix $\mathbf{R}_\omega$ transforms the vector space $\mathbb{R}^{n+1}$ as in equation (16). But in binary form, this transformation acts as a Boolean function on $(n+1)m$ bits, and it retrieves the result of an alternative computation. This doesn't seem to have been effectively performed, since all we did was to verify a case which can't solve the problem, but it was performed in parallel, in an encrypted way. Therefore, this experiment proves the existence of alternative computations.

The reader may object that, by applying $\mathbf{R}_\omega$ defined as in equation (24), what we do in fact is to reverse the algorithm for ω_0 to the initial state, and then to apply it again for ω. So, one may say, it's no surprise that it works. Indeed, it's true that $\mathbf{R}_\omega$ can be expressed as composed of these transformations. But we applied it in its direct form, not decomposed into undoing-redoing the computation, and it indeed reinterprets the result for ω_0 to find the result for ω. Since $\mathbf{U}$ is invertible, undoing the computation and doing it again for a different ω is equivalent to obtaining the result by applying $\mathbf{R}_\omega$ directly on the result for ω_0. The conclusion of the previous sections, that other computations take place in parallel, hidden from the user interface, is verified experimentally.

Remark 2. This example may seem to be an exceptional situation, built specifically to allow the recovery of the alternative results. It is indeed a special example, but only by being designed so that we, humans, can follow it, despite of our limited computational power. But there are such transformations for all programs covered by Theorem 1. □

Experiment 1 (The Partition Problem Experiment)**.** The reader can download the `C++` source code of a program that does all of these from (Stoica, 2023c). The source code can be verified, and the program can be executed to see that it is indeed possible to extract the results of all alternative computations from the execution of a single computation.

The readers can use the debugger to verify the program, or they can write their own versions of the program.

Remark 3. While we use a decryption key $\mathbf{R}_\omega$ that is different from the encryption key $\widetilde{\mathbf{S}}_\omega$, these are parts of the same overall transformation, just like $\mathcal{R}_0$ and $\mathcal{R}_N$ from the proof of Theorem 1 are both parts of the transformation $\mathcal{R}$. The encryption key represents the part of the transformation for states in which the clock bits encode the time $t = 0$, and the decryption key represents the part of the transformation for states in which the clock bits encode the time $t = N$. But since, for simplicity, we didn't include the clock bits in the equations, the two parts seem to be distinct transformations. □

Remark 4 (Fun Fact: P *vs.* NP)**.** The partition problem is *NP-complete*. The NP-complete problems are important for the famous *P versus NP problem*, one of the seven *Millennium Prize Problems*. The Clay Mathematics

Institute offers a US$ 1 000 000 bounty for the solution of each of these seven problems.

But what is the *P versus NP problem*?

A problem is of class P (for "polynomial time") if it can be solved by an algorithm in a *polynomial time*, that is, the time needed to solve it increases at most as a power of the size of the input data. A problem is of class NP (for "nondeterministic polynomial time") if, given a candidate solution to the problem, there is an algorithm that can verify in a polynomial time if the candidate solution is a solution. It is called "nondeterministic" because a nondeterministic Turing machine can guess a correct solution by luck, and then verify it in a polynomial time.

The *P versus NP problem* asks whether the two classes of problems P and NP are the same or are different.

There is a special class of NP problems, called *NP-complete*. Showing that any one of these problems can be solved in a polynomial time, implies that all other NP problems can. This would be a proof that P=NP, so it would solve the *P versus NP problem*!

The partition problem is NP-complete, and it is considered to be "the easiest hard problem" (Hayes, 2002). The algorithm presented in this Section is not the simplest one, but it still requires only a polynomial time in the size of the input data.

We have seen that even when a single partition is verified by the algorithm I presented, all other partitions are verified in parallel. So all partitions are verified in parallel, in a polynomial time. Does this mean that the algorithm that I presented proves that P=NP? Not at all, because to actually access the results of all 2^n computations taking place in parallel, we need to apply the appropriate transformation $\mathbf{R}_\omega$ for each of them. So we need a time proportional to 2^n, which is not polynomial. □

5 "Think of a number" and other implications

Using the Partition Problem Experiment from Section §4, let us derive some implications of Theorem 1. To prove these implications, I will use thought experiments. These experiments are totally feasible in principle, but since the practical difficulties are great, we will perform them only in our minds.

Implication 1. The user interface is not sufficient to fix a unique interpretation of a computation.

Thought Experiment 1 (Proving Implication 1)**.** Consider a server (which is a computer) providing the following service, based on the program from the Partition Problem Experiment. The service works as follows:

1. A client sends to the server a partition $\omega = (w_1, \ldots, w_n)^\intercal \in \{+1, -1\}^n$.
2. The server verifies a partition ω for a fixed list of numbers $\mathsf{v} = (x_1, \ldots, x_n)^\intercal \in \mathbb{Z}^n$ as in equation (11). This is the only list of numbers for which the service is offered.
3. The server returns the result to the client.
4. But, in addition, it offers the client the option to encrypt the partition, by using a simple `XOR` cipher. The clients receive a program that generates a personalized key. The key is translated into a matrix $\mathbf{S}_\omega$ that has only ±1 on the diagonal, all its other cells being 0. The client program also computes the decryption key, the matrix $\mathbf{R}_\omega$ from equation (24), and applies it to decrypt the result received from the server.

Suppose that the number of requests is very high, and the server's administrator decides to optimize in the following way. The requests are collected, and once a day, the results are computed for each encrypted partition in the list of requests, only once. So even if a partition appears, after encryption, a hundred times in the list of requests, the result is computed once, and sent back to the hundred users.

In one lucky day, 2^n identical requests are received, and they are verified only once, of course. These are encrypted requests, and the actual partitions the clients wanted to be verified are different: there are 2^n distinct partitions. And yet, the server verified them all at once, since they have the same encrypted form. Each client sees in the user interface of the client program their unencrypted partitions, and then the corresponding results.

We obtained a situation in which the user interface doesn't fix a unique interpretation, but rather each of the possible 2^n interpretations is fixed by a different user interface. □

Implication 2. A computer cannot think of a number unambiguously.

Thought Experiment 2 (Proving Implication 2)**.** This starts with the set up from Thought Experiment 1, with the additions that

- The server runs an AI that administers the requests.

- The AI adds a personal note about the resulting numbers, something that those numbers triggered it to think about. For example "Seven reminds me of Snow White", "Three reminds me of D'Artagnan", "Ten reminds me of the story of the ten men crossing a river" *etc.*

- The decryption key takes into account all the ramifications of the AI thinking about the number.

In more details, the AI receives the requests, runs the program to verify the partition as in Thought Experiment 1, and sends to the client the final state, including the personal note. So now, to decrypt the message, the client needs a key that decrypts more than just the result of the program that verifies the partition, but the full final state of the computation, including the AI's personal note. Now the computation affects more than the memory strictly needed for the result, it can affect potentially any bit of memory of the server. As shown by Theorem 1, such a decryption key exists, but since it has to decrypt all the bits from the server, it is very large and very difficult to compute.

Luckily, this is a thought experiment, so we don't have to build the AI and compute the key. It is sufficient to show that such a key exists in principle, and this was done by Theorem 1. The server's transitions are described by a function $\mathcal{F}$ as in equation (3). Its memory already contains an AI waiting to collect the requests and to compute the result. When the AI starts the program to verify the partition sent by multiple clients, the server is in a particular initial state. The states corresponding to different possible partitions are all distinct. Since the input data are saved until the request is completed, and since the server also has a clock, the conditions for Theorem 1 are satisfied. So the possible final states of the server after the computation, at the same time N, are also distinct. So the decryption key must be computed based on the full encryption key, seen as a function that changes the relevant bits and leaves the other bits on the server unchanged. This function has the matrix form

$$\mathbf{S}_{\omega,\text{total}} = \begin{pmatrix} \widetilde{\mathbf{S}}_\omega & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}, \tag{26}$$

with $\widetilde{\mathbf{S}}_\omega$ from equation (19), to change only the relevant bits for the input.

Then, the decryption key is, as follows from Figure 4,

$$\mathbf{R}_{\omega,\text{total}} = \mathcal{F}^N|_{\mathcal{S}_0} \mathbf{S}_{\omega,\text{total}} (\mathcal{F}^N|_{\mathcal{S}_0})^{-1}, \tag{27}$$

where since the function

$$\mathcal{F}^N|_{\mathcal{S}_0} : \mathcal{S}_0 \to \mathcal{S}_N, \tag{28}$$

is bijective, it is invertible, so $\left(\mathcal{F}^N|_{\mathcal{S}_0}\right)^{-1}$ exists.

This means that all possible histories in which the AI computes the result and lets its mind wandering about it happen simultaneously. Each of them is in an encrypted form with a different key, but by Thesis CST′ they are equally authentic computations. And, given the number of bit flips used in the design of the logic circuits, what computation isn't encrypted?

So there is no unique, objective fact that the computer thinks of a particular number. At the same time, the AI can be equally interpreted as thinking of other numbers or of something totally different, and its parallel minds are wandering in completely different ways. □

Remark 5. If Laplace's demon himself would know the state of the server down to every particle, he would still be unable to read unambiguously the number the AI is thinking about, nor the musings of the AI triggered by that number. □

Implication 3. The AI's past history can be "retroactively" erased, replaced, and restored.

Thought Experiment 3 (Proving Implication 3)**.** Suppose that a client uses the server from the Thought Experiment 2, to compute the result for the partition ω. But then, for whatever reason, instead of decrypting the result by using the key decryption key $\mathbf{R}_{\omega,\text{total}}$, it decrypts it by using another decryption key $\mathbf{R}_{\omega',\text{total}}$, corresponding to another partition $\omega' \neq \omega$. The decryption of the result will, of course, correspond to the computation done for ω', as in Figure 7. It will contain completely different musings of the AI, as if this was what happened all along.

Question 3. *Did the server implement the AI that computed the result for ω_0, or the AI that computed it for ω, or the AI that computed it for ω'? If it was the AI that computed only the result for ω_0 or ω, where did it go? Where did the AI that computed the result for ω' come from?*

Apparently, the history of the AI, with all its reported thoughts and experiences, was erased and replaced with another one. The thoughts of the AI in the time interval between encryption and decryption are undefined, because the encryption predicts different thoughts than those revealed by the decryption.

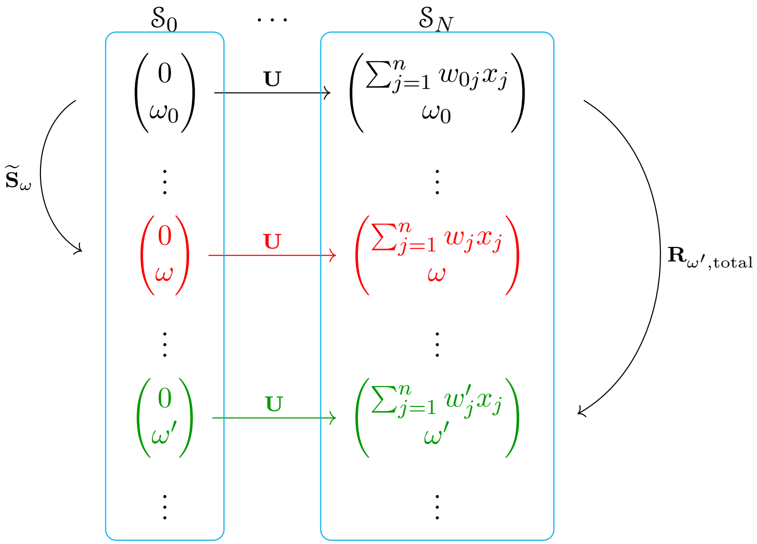

$\mathcal{S}_0$
$\cdots$
$\mathcal{S}_N$
$\begin{pmatrix}0\\ \omega_0\end{pmatrix}$
$\mathbf{U}$
$\begin{pmatrix}\sum_{j=1}^n w_{0j}x_j\\ \omega_0\end{pmatrix}$
$\widetilde{\mathbf{S}}_\omega$
$\begin{pmatrix}0\\ \omega\end{pmatrix}$
$\mathbf{U}$
$\begin{pmatrix}\sum_{j=1}^n w_j x_j\\ \omega\end{pmatrix}$
$\mathbf{R}_{\omega',\text{total}}$
$\begin{pmatrix}0\\ \omega'\end{pmatrix}$
$\mathbf{U}$
$\begin{pmatrix}\sum_{j=1}^n w'_j x_j\\ \omega'\end{pmatrix}$

Figure 7: Changing the decryption key erases the already happened history of the AI corresponding to the computation for ω and replaces it retroactively with a history corresponding to the computation for ω'.

Note that while the past history that we attribute to the AI is erased and replaced, the computer's physical history remains unchanged. □

Remark 6. An implication of Thought Experiments 1 and 3 is that we can connect differently the user interface to the computer to access, in principle, any alternative computation. Remember that the user interface serves to fix a convention, which is then followed by the hardware and software manufacturers, and results in maintaining the illusion of an unambiguous computation. Now suppose that the keyboard has an encryption key, and it sends encrypted messages to the computer. Similarly, the display and the audio devices receive encrypted data from the computer and decrypts them. We assume the encryption/decryption scheme used in the previous Thought Experiments. If the decryption key is replaced with one corresponding to another encryption key, the user will have the illusion that it accessed a version of the computer from an alternative reality. □

Implication 4. An AI trained in the history of our world can appear as being trained in the history of another logically possible world. For example, the AI can appear as being trained in the history of the Star Wars universe, the Star Trek universe, the Harry Potter universe, the Lord of the Rings universe, the Marvel universe, the DC universe *etc.*

Thought Experiment 4 (Proving Implication 4)**.** We assume the conditions of Theorem 1. Consider a powerful computer whose initial state contains an AI and the training data about the history of our world. The computer is left running until the AI is trained well enough on the data.

Then, suppose we access the final state by using a decryption similar to that from equation (27), but corresponding to a different initial state, containing in the training data not the history of our world, but the history of a fictional universe. The memory containing the history of our world can be changed, by flipping bits accordingly, to obtain such an alternative history. Therefore, there is a decryption key, albeit extremely large and complicated, that decodes an AI trained with the alternative fictional history of our choice from the AI trained with the history of our world. □

6 Analog and quantum computations

The proof of Theorem 1 and its implications didn't make use of discreteness in an essential way. These results can be extended immediately to any dynamical system, and any implementation consistent with Principles CSP1 and CSP2.

The main change compared to Definition 1 of discrete dynamical systems is that time can be a continuous parameter $t \in [0, +\infty)$ or $t \in \mathbb{R}$. In this case, the dynamical law μ specifies how the system transitions from a state s to a state $\mu^t(s)$ after the time interval t, if no external cause affects the system. This is given by a family of functions $\{\mu^t : \mathcal{S} \to \mathcal{S} | t \in [0, +\infty)\}$ parametrized by all positive durations, so that for any $s \in \mathcal{S}$ and any $t, t' \in [0, +\infty)$,

$$\begin{aligned} \mu^0(s) &= s \\ \mu^t(\mu^{t'}(s)) &= \mu^{t+t'}(s). \end{aligned} \tag{29}$$

To obtain maximal ambiguity, we also need to make sure that our system satisfies the conditions of Theorem 1. It is sufficient that the histories corresponding to distinct inputs remain distinct during the computation. Then, the same proof as for Theorem 1 can be used for obtaining the result.

Theorem 2. *Theorem 1 generalizes to all dynamical systems whose histories corresponding to distinct initial states don't cross each other, and which don't return to previous states during the considered time interval.*

Proof. Instead of a permutation of the initial set $\mathcal{S}_0$, we use any bijective function from $\mathcal{S}_0$ to $\mathcal{S}_0$. For any two possible initial states, there are numerous bijective functions that swap those states. If $\mathcal{S}_0$ has a finite number k of elements, this number is $(k-2)!$. If $\mathcal{S}_0$ is infinite, this number is infinite. This entails the existence of reinterpretations in which any computation with the algorithm that the system implements is interpreted as any other of its computations. □

Also, all the implications from Section §5 follow.

Let us see what kinds of computing systems these results generalize to.

Example 1 (Analog computers)**.** An analog computing system is a dynamical system with continuous time $\mathbb{R}$ or $[0,\infty)$, and its set of possible states $\mathcal{S}$ require in their descriptions continuous parameters, although it may have discrete parameters as well. Then, any continuous reparametrization $f : \mathcal{S} \to \mathcal{S}$ can be used as the reinterpretation from the proof. □

In particular the results extend to analog models of neural networks.

Example 2 (Quantum systems)**.** Theorem 1 and its implications also extend to quantum systems, for example quantum computers. In this case the state space $\mathcal{S}$ is a complex vector space. The dynamical law is represented by a one-parameter unitary group $\left(\widehat{\mathbf{U}}_t : \mathcal{S} \to \mathcal{S}\right)_{t\in\mathbb{R}}$, possibly alternated with projections. For a quantum computer the dynamics is unitary during the computation, so distinct state vectors are mapped into distinct state vectors after the same time interval. Then, a reinterpretation simply corresponds to a change of the basis in the vector space $\mathcal{S}$, and the proof goes just like that of Theorem 1. The only important difference is that, at the end of the computation, the state vector is measured, which entails a projection on a particular set of orthogonal subspaces of $\mathcal{S}$.

But the result applies even if the dynamics includes wavefunction collapses, if the input is preserved and the system changes without returning to a previous state. □

Corollary 1. If the human mind is just a computation implemented by the brain, it suffers from the ambiguity from Theorem 2.

Proof. For the brain to implement the mind as a computation, the implementation has to satisfy Principles CSP1 and CSP2.

The human body, including the brain, is a physical system. It holds memories of past events and it changes in time irreversibly, so it satisfies the conditions of Theorem 2. Neurons seem to work as analog systems, in which case the ambiguity follows from Example 1. If the existence of mind requires quantum effects, including the wavefunction collapse, the ambiguity follows from Example 2. □

7 Crucial Experiment: Are you a computation?

We have seen in Corollary 1 that if the mind were reducible to a computation, the same physical process of the brain would implement many contradictory minds at once. Therefore, only one of the following can be true: Option 1, that the same computation supports many contradictory minds at once, or Option 2, that mind is not reducible to a computation.

Question 4. *Is it possible to make a* crucial experiment*, i.e. an experiment able to eliminate either Option 1 or Option 2?*

Yes, and now I will show that your mind doesn't reduce to a computation. Not only it doesn't reduce to a discrete computation, but also neither to an analog or a quantum computation that satisfies both Principles CSP1 and CSP2. I apologize for making it about the reader's mind, but I don't know how to show it for the minds of other people.

For this experiment you will not need to leave your armchair and go to a laboratory. It will all take place in your mind, but it is not just a thought experiment, because you can do it effectively. All it takes is to go through several steps of reasoning and inquiry.

Experiment 2 (Experimentum crucis)**.**

Step 2.1. The prerequisite for this experiment is to understand what I showed in this article up to this point. If necessary, please review it.

Step 2.2. Remember Corollary 1: if your mind would be reducible to computation, the same physical state of the brain should support as many possible minds as the number of possible states in which the brain can be.

Step 2.3. Remember Implication 4: if your mind would be reducible to computation, your brain should support at the same time alternative minds holding any possible worldview.

Step 2.4. Your mind holds a worldview. This worldview is reasonably consistent. You may believe some contradictory things simultaneously (for example that a convention we made about the bits in a computer can be an objective mind). We all hold a few inconsistencies in our minds, we usually maintain them under control and manage to convince ourselves that there is no inconsistency. But these inconsistencies are usually not as blatant, you could hold worse inconsistencies, such as believing that you live on the tenth floor of a single floor house, or that your father is your daughter, or that you were born thousands of years ago or thousands of years from now. Or like the inconsistencies in a dream, where you step through the door of your house and you find yourself in your childhood house, and where people around you morph into other people or a text written on a piece of paper changes while you look away.

Step 2.5. On the other hand, as seen in Implication 4, the alternative computations implemented by the same physical process can contain any sort of data. This data may be worldviews that are apparently about facts, but are extremely inconsistent, like in the wildest dream. It may be data that doesn't even make sense, disparate images and words that can't be integrated consistently in a larger view. It may even be pure noise. And the number of such possible alternative minds increases exponentially with the number of inconsistencies of their content, as illustrated in Figure 8.

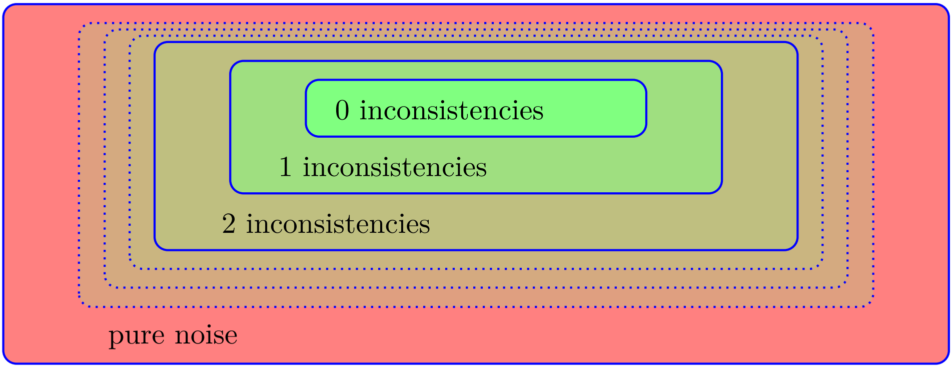


Figure 8: Worldviews classified by the level of inconsistency.

Step 2.6. Let us now restrict our attention to the possible minds that read and understood this article, including this Experiment, and which may contain inconsistencies about other things. Some of these possible minds are relatively consistent. More of them contain a few inconsistencies about other things. Many more contain more inconsistencies about other things. The more inconsistencies they contain, the more such possible minds there are. So alternative minds like yours form an exponentially small minority among all alternative minds, even if we count only those that contain the memory of reading this article.

Step 2.7. Now you will give a ballpark estimate of the probability that, if your brain supports all these minds with their own worldviews as in Step 2.6, your mind is one of the few consistent minds. Remember that each mind with a particular worldview can be reinterpreted as another mind with any other worldview. Therefore, the probability distribution of these possible alternative minds is invariant under automorphisms of the dynamical system called brain. Hence, even if some of the minds would be favored, for example by natural selection, all possible minds are uniformly distributed with respect to the measure on the space of states of a physical system like your brain. This means that no possible mind is favored by the probabilities. Since the number of possible minds with very inconsistent worldviews as in Step 2.6 is much larger than the number of possible minds with reasonably consistent worldviews like yours, the probability that the minds with unreasonable worldviews really exist is vanishingly small. They do exist as simulations, as alternative computations, but not as genuine minds like yours. If they were genuine minds, to have a mind like yours when the probability to have an inconsistent mind is extremely close to 1, you'd had to be extraordinarily lucky. □

Therefore, while a simulation of a computation is a computation,

Conclusion 1. A simulation of a mind is not a mind.

Now I will try to anticipate and address several possible objections to the conclusion of Experiment 2.

Objection 3. Experiment 2 is an introspection, so it can't be trusted.

Reply 3. Experiment 2 consists of steps of reasoning and inquiry, which are no different from what we do about anything else. When we verify the proof of a mathematical theorem, we make similar steps, and rely on the content of our mind, on the memory of previous readings and calculations we did in the past, and we assume that the content of our mind is consistent. If Experiment 2 is introspection, then so is any scientific research. □

Objection 4. A probabilistic proof is not a proof.

Reply 4. All experiments that confirm theoretical predictions are in fact probabilistic. For example, the discovery of a new particle like the Higgs boson is never 100% certain. With 5 sigma, there is still a chance in 3.5 million that, given that the Higgs boson doesn't exist, CERN got a false positive. □

Objection 5. A mind that is too inconsistent can't survive, so such minds are excluded by natural selection. So it isn't fair to count very inconsistent minds among the possible alternative minds.

Reply 5. If the mind would be reducible to a computation, every such mind can be reinterpreted as numerous alternative minds that are extremely inconsistent. They are implemented by the same organism that survived the natural selection, but these alternative minds need not have the ability to survive. It is sufficient that the organism implementing them has the ability to survive, and they will survive too. □

Objection 6. The argument from Experiment 2 suffers from selection bias. Only a mind that can read this article can perform this experiment. The alternative computations that are totally inconsistent or pure noise are automatically excluded.

Reply 6. As described in Step 2.6, an alternative mind may contain memories of reading this article and Experiment 2, and be very inconsistent about many other things. Therefore, even the possible inconsistent minds having representations of this very article would overwhelmingly dominate the real consistent minds that actually read it. This allows the readers to estimate for themselves if they are among the very unlikely minds, or among the highly likely but inconsistent minds. □

Remark 7. Experiment 2 is personal precisely in order to avoid selection bias. You can ask other people to do it for themselves, and they will tell you the result, but you can't infer from this that the brains of other people don't contain minds with all possible worldviews. The reason is that, even if it would, you are using a particular user interface established by social interactions, and you will interact only with minds able to give an answer through that interface. And that mind is the one able to survive in this world. This is why for this experiment we can only rely on our own mind, and not on the reports of other people, because we can reach only a mind that is sufficiently consistent to fit in this world.

This problem is similar to the *problem of other minds* (Avramides, 2020), but in this case you know your mind is not reducible to a computation, but you can't verify this for other people's minds. But, as in the case of the problem of other minds, a totally skeptical position leads to a variant of solipsism. If we agree that solipsism is not a tenable position, we should agree as well that other people's minds are not reducible to computation.

A way to accept the possibility that also other people's minds are not reducible to computations is the following. Since our mind is not reducible to a computation, it requires the violation of Principle CSP1 or Principle CSP2 or maybe both. If our brains can violate one of these principles, it makes sense to assume that other people's brains can too.

Another way is to realize that if other people's minds are reducible to computations, when you estimate the probabilities for consistent and inconsistent minds, you should take into account that you could be one of the inconsistent alternative minds of somebody else, or even of any sufficiently complex computer. □

Objection 7. If we follow the argument from Remark 7, shouldn't we accept as well that an AI mind, who made the experiment is as real as other people's minds?

Reply 7. According to Remark 7, the only reason we accept other people's minds as real is because we expect their brains to violate Principle CSP1 or Principle CSP2 or maybe both, like we concluded our own brain does. But the AI is a computation, by design, and so it satisfies both Principles CSP1 and CSP2. If the AI would have a mind like the human mind, it would have to violate at least one of them.

But Bayes' theorem gives a stronger reason why you are not an AI.

Let H be the hypothesis that the mind is reducible to computation. Let $P(H)$ be your initial credence that mind can be reduced to computation. Then your initial credence that mind can't be reduced to computation, is $P(\neg H) = 1 - P(H)$. (At this point I have nothing more to offer than your own credence, since science says nothing about the validity of H.)

If mind isn't reducible to computation, it collects its data from the real world, so the data are facts, and they are consistent (even if from the consistent data sometimes we draw inconsistent conclusions).

Let C be the event that an interpretation of a computing system contains consistent data. The probability that a mind is consistent given that it is reducible to a computation is $P(C|H)$, and the probability that a mind is consistent given that it is irreducible to a computation is $P(C|\neg H)$. So the total probability that you'll find your mind to be consistent is

$$\begin{aligned} P(C) &= P(C\&H) + P(C\&\neg H) \\ &= P(C|H)P(H) + P(C|\neg H)P(\neg H), \end{aligned} \tag{30}$$

where $P(C\&H)$ is the probability that C happened and H is true, and $P(C\&\neg H)$ is the probability that C happened and H is false.

At the same time, the probability that a reinterpretation of a computation that simulates a mind is consistent is exponentially small in the system's size, *i.e.* something like

$$P(C|H) \approx 0.000 \ \ldots\ldots 001 \tag{31}$$

with a huge number of zeroes before the first significant digit.

If the computing system is analog or quantum, $P(C|H)$ may seem to be 0, but we should more fairly discretize it into distinguishable states.

Let's put all of these together.

$$\begin{aligned} P(H|C) &= \frac{P(C|H)P(H)}{P(C)} && \text{[Bayes' theorem]} \\ &= \frac{P(C|H)P(H)}{P(C|H)P(H) + P(C|\neg H)P(\neg H)} && \text{[equation (30)]} \\ &= \frac{P(C|H)P(H)}{P(C|H)P(H) + P(\neg H)} && [P(C|\neg H) = 1] \\ &= \frac{0.000 \ \ldots\ldots 001}{0.000 \ \ldots\ldots 001 + P(\neg H)} && \text{[equation (31)]} \end{aligned} \tag{32}$$

So, even if $P(\neg H)$, your prior credence that the mind is irreducible to a computation, was as small as the probability that a random reinterpretation is consistent from (31), your credence that the mind is reducible to a computation should now be updated to 50/50!

But if your prior credence that the mind is irreducible to a computation was 0.5, your credence that the mind is reducible to a computation should now be updated to something like

$$0.000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ldots 001!$$

Still, if you were absolutely 100% certain that the mind is reducible to a computation, you get to keep your certainty, because

$$\begin{aligned} P(H|C) &= \frac{0.000 \ \ldots\ldots 001}{0.000 \ \ldots\ldots 001 + P(\neg H)} \\ &= \frac{0.000 \ \ldots\ldots 001}{0.000 \ \ldots\ldots 001 + 0} \\ &= 1 \end{aligned} \tag{33}$$

Bayes' theorem allows you to do this. Similarly, a consistent computation that simulates a mind can apply the reasoning I presented and conclude that is itself irreducible to a computation. This would be a false positive! A "computational zombie"! □

8 Discussion

There is increasing support among the specialists for the idea that an AI can be sentient in the same way as humans can. This is favored by the fact that more and more powerful AIs are developed, and there seems to be no human artistic and intellectual activity that can't be simulated, at least in principle, by an AI. And I won't deny this. But Theorem 1 shows that there is a very high ambiguity in the content of the AI's mind. A computer can't think unambiguously even of a number, as shown in Implication 2. And there is no objective reason why one of the numbers the computer is thinking about is more special than others.

The experiment from Section §7 shows that we are not reducible to computations. But this can be inferred only by the individuals who perform the experiment, for themselves. For such an individual it should follow that the mind can't be simulated on a computer that satisfies the Principles of Computer Science. The AIs are just simulations, not minds. They are "computational zombies" so to speak.

In the literature it is often proposed that computation is something that in fact violates the principles of Computer Science, particularly structure independence. This was proposed by various authors to escape the ambiguity of computation already noticed in the triviality argument (see Appendix §A). But if computation would violate Principle CSP2, computers as we know them wouldn't be possible. Maybe the mind requires such a violation, but then it can't be implemented on the current computers, which work according to the principles of Computer Science, in particular Principle CSP2.

This doesn't mean that there are no physical systems that can think. Humans are such an example, and I see no reason why it would be impossible to artificially build other beings able to think. But the results from this article show that, for a machine to think, it should do more than merely simulate thinking. And I don't know what.

In a future article I will investigate proposals made in various theories of mind, that mind reduces to structure or function, down to the finest organizational levels of matter (Stoica, 2023a,b).

A Computer's ambiguity and triviality arguments

Is Theorem 1 a variant of the *triviality argument*?

In its simplest form, the triviality argument claims that almost every physical system implements every finite state machine (Lycan, 1981; Putnam, 1988; Searle, 1990; Egan, 1994).

Putnam presented it in the form of a Theorem (Putnam, 1988, p. 121)

> Every ordinary open system is a realization of every abstract finite automaton.

Putnam showed that an open system whose history doesn't return to a previous state during a time interval $[0, \mathcal{T}]$ implements, in that interval, all computations done by any finite state machine (FSM) without inputs and outputs. Since Putnam's proof takes three pages, let me sketch a simpler proof, but for a sufficiently isolated physical system $(\mathcal{S}, \mu, \mathbb{R})$.

Proof. Let $S_0 \mapsto \ldots \mapsto S_N$ be a computation of a FSM, to be shown to be implemented by the physical system. A FSM is just a discrete dynamical system with a finite number of states. The computation $S_0 \mapsto \ldots \mapsto S_N$ is itself a FSM with a single computation, and its states can repeat. Without loss of generality, we can choose $\mathcal{T}$ so that $N \leq \mathcal{T} < N+1$, *e.g.* by changing the measurement unit for time. Let $f : [0, \mathcal{T}] \to \mathcal{S}$ represent the state of the physical system as a function of time. Recall that $f([0, \mathcal{T}])$ is a set of states from $\mathcal{S}$, the *image* of the function f. Since during the time interval $[0, \mathcal{T}]$ the physical system doesn't return to a previous state, for every $s \in f([0, \mathcal{T}])$ there is a unique time $t \in [0, \mathcal{T}]$ so that $f(t) = s$, denoted as $t = f^{-1}(s)$. Define the function $\alpha : f([0, \mathcal{T}]) \to \{S_1, \ldots, S_N\}$,

$$\alpha(s) := S_{\lfloor f^{-1}(s) \rfloor}, \tag{34}$$

where $\lfloor t \rfloor$ is the integer part of t. Then, α realizes an implementation of the computation $S_0 \mapsto \ldots \mapsto S_N$. □

The mapping is not one-to-one, since different states of the physical system will be mapped into the same state of the FSM. But this is all right, because any physical computing system has the same property. For example, a memory cell in a computer may be charged or discharged, but it is never 100% or 0% charged, so the same bit is represented by a continuum of possible physical states. This should also be expected because of the multiple realizability of the computing systems.

The implementation of a single computation by a physical system is called the *simple mapping account* of computation (Godfrey-Smith, 2009). It was argued that a computation is not really a computation, unless the system implements the entire algorithm, and it contains the computations corresponding to other possible initial data. Such computations that didn't happen but could have happened if the initial data were different are called *counterfactual*. This requirement of counterfactuality seems artificial and problematic: why would a system do a computation, or be conscious, only if there are alternative possible computations, even if they are never realized? Does it have the "superpower of knowing the alternative histories", even though they didn't happen? But anyway the proof of Theorem 1 satisfies counterfactuality.

At any rate, the triviality argument was extended to include counterfactuals. One may choose a property of the physical system that is constant along a history, but can be different for different histories of the system. It could be a conserved quantity. Chalmers argued that a system that doesn't return to a previous state and has a conserved property like this, that can take sufficiently many distinct values, can implement all computations done by any FSM (Chalmers, 1996). The proof is similar to Putnam's, all we have to do is to highlight sufficiently many alternative histories of the physical system and map them to the computations of the FSM. In the proof of Theorem 1, the conserved property is the input data themselves.

Godfrey-Smith improved the triviality argument for FSMs with inputs and outputs by arguing that, by changing the *transducer layer* (the interface between the system and its environment) of a physical system implementing a FSM, this can exhibit a different FSM within the same physical system (Godfrey-Smith, 2009). He argued that every sufficiently complex system can be turned into a computing system with the same computations as a human conscious being by connecting it properly to an appropriate (human-like) transducer layer. Moreover, he argued that using different transducer layers will get different human-like conscious beings. He concluded that functionalism in its common form implies that "Any sufficiently complex system has non-marginal mental properties." He argued that strengthening the constraints in the functional approaches would help avoid this triviality. But there is no proof that this is sufficient to achieve uniqueness.

More often the triviality argument is simply ignored. According to Godfrey-Smith (Godfrey-Smith, 2009, p. 274),

> Given the threat that such arguments pose, it is surprising how

> little they have been discussed, especially as the mainstream functionalist literature does not use accounts of the realization of functional structures that make it clear that triviality problems do not arise. Many accounts of realization used by functionalists are so schematic that it is uncertain how these problems are handled.

But, as modest as it was, the debate that took place was important in shaping and extending the triviality argument.

Nevertheless, let us face a potential objection.

Objection 8. How do Theorems 1 and 2 address the counterfactuality objection against the triviality argument?

Reply 8. If the requirement of counterfactuality matters, since Theorems 1 and 2 establish an automorphism of the computing system for all possible inputs of the implemented algorithm, they include the counterfactuals. The proof is based precisely on reinterpreting as counterfactual histories what the convention behind the user interface sells to us as "factual history". □

Another objection to the triviality argument was that the FSMs involved are not general, because they don't have inputs and outputs.

Objection 9. In Theorems 1 and 2, the computations don't have inputs and outputs.

Reply 9. They are about computers, and computers have both inputs and outputs. While the proof deals with a situation when the computer carries out the computation without taking new inputs, this should not be relevant, since if we believe that a computer can think, we should also believe that it is able to think between inputs as well.

Nevertheless, if one considers this so relevant, we can make the computer take inputs during the time relevant to the proof by pre-recording them in its memory. A simple example is to use a computer program that allows the user to record a *macro*. The user starts the record, then performs various operations, which are inputs, and at the end stops the record. Then, the user can play the recorded macro, and it will do the same operations again. The user can, in principle, record very sophisticated commands, say all the input commands given during a full day. Then, the next day, the user can make the computer do the same as in the previous day, *as if the user inputs data and commands in real time*, but without actually providing those inputs again. And the computer can of course display the outputs. Therefore, the existence or absence of inputs and outputs doesn't affect at all the results. □

Suppose that we add more restrictions to what counts as a computation. For example, Chalmers attempted to avoid the triviality argument by requiring the states of the computing systems to have more structure, so that each state of both the computation and the implementation to consist of more components (Chalmers, 1994, 1996, 2011). Even with this constraint, he admitted the following "Can a given system implement more than one computation? Yes." (Chalmers, 2011, p. 334). Another proposal is the *mechanistic account* of computation, in which the properties of the system matter, even though they can be implemented by multiple different systems (Piccinini, 2015). Such proposals are referred to as "accounts of computation" (Rescorla, 2020; Colombo and Piccinini, 2023), although, as we have seen, they are not accounts of what a computation is in Computer Science.

Unless such additional constraints contradict Computer Science, they can't contradict Theorem 1 and its implications. Reinterpretations like those from the proof are used practically everywhere in the hardware manufacturing to optimize the circuits, in the software development, in cryptography, and of course in the very notions of Turing equivalence and universality. Therefore, even if one may argue that Theorems 1 and 2 and the implications are variants of the triviality argument, they overlap perfectly on what computers currently do.

I will discuss the option of violating Principle CSP2 in the articles (Stoica, 2023a,b).

A.1 Acknowledgments

The author thanks Justin Sampson, Jonathan Mason, Cris Calude, and Paul Tappenden for helpful discussions. Nevertheless, the author bears full responsibility for the article.

References

Avramides, A. (2020). Other Minds. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2020 edition.

Chalmers, D. (1994). On implementing a computation. *Minds and Machines*, 4(4):391–402.

Chalmers, D. (1996). Does a rock implement every finite-state automaton? *Synthese*, 108(3):309–333.

Chalmers, D. J. (2011). A computational foundation for the study of cognition. *Journal of Cognitive Science*, 12(4):325–359.

Colombo, M. and Piccinini, G. (2023). *The Computational Theory of Mind.* Cambridge University Press, New York.

Cook, M. (2004). Universality in elementary cellular automata. *Complex systems*, 15(1):1–40.

Davis, M. (2004). *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions.* Courier Corporation.

Egan, G. (1994). *Permutation city.* HarperCollins/Eos, New York, U.S.

Fredkin, E. and Toffoli, T. (1982). Conservative logic. *Int. J. Theor. Phys.*, 21(3-4):219–253.

Gandy, R. (1980). Church's thesis and principles for mechanisms. In Barwise, J., Keisler, H., and Kuhnen, K., editors, *The Kleene Symposium*, volume 101, pages 123–148. Elsevier, Amsterdam, North-Holland.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game 'Life'. *Sc. Am.*, 223:20–123.

Gentry, C. (2009). *A fully homomorphic encryption scheme.* Stanford University, Stanford.

Giunti, M. (1997). *Computation, dynamics, and cognition.* Oxford University Press.

Godfrey-Smith, P. (2009). Triviality arguments against functionalism. *Philosophical Studies*, 145(2):273–295.

Golomb, S. (1959). On the classification of Boolean functions. *IRE transactions on circuit theory*, 6(5):176–186.

Hayes, B. (2002). The easiest hard problem. *American Scientist*, 90(2):113–117.

Kudlek, M. and Rogozhin, Y. (2002). A universal Turing machine with 3 states and 9 symbols. *Lecture notes in computer science*, pages 311–318.

Lycan, W. (1981). Form, function, and feel. *J. Philos*, 78(1):24–50.

Marcolla, C., Sucasas, V., Manzano, M., Bassoli, R., Fitzek, F. H., and Aaraj, N. (2022). Survey on fully homomorphic encryption, theory, and applications. *Proceedings of the IEEE*, 110(10):1572–1609.

Neary, T. and Woods, D. (2009). Four small universal turing machines. *Fundamenta Informaticae*, 91(1):123–144.

Nielsen, M. and Chuang, I. (2010). *Quantum Computation and Quantum Information.* Cambridge University Press.

Piccinini, G. (2015). *Physical computation: A mechanistic account.* OUP, Oxford.

Putnam, H. (1988). *Representation and reality.* MIT press.

Rescorla, M. (2020). The Computational Theory of Mind. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, fall 2020 edition.

Rogozhin, Y. (1996). Small universal Turing machines. *Theor. Comput. Sci.*, 168(2):215–240.

Searle, J. R. (1990). Is the brain a digital computer? In *Proc Am Philos Soc*, volume 64, pages 21–37. JSTOR.

Sieg, W. (2002). Calculations by man and machine: Conceptual analysis. In Sieg, W., Sommer, R., and Talcott, C., editors, *Reflections on the foundations of mathematics (essays in honor of Solomon Feferman)*, volume 15, pages 387–406. AK Peters/CRC Press.

Slepian, D. (1953). On the number of symmetry types of Boolean functions of n variables. *Canad. J. Math.*, 5:185–193.

Stoica, O. C. (2023a). Are observers reducible to structures? *Preprint arXiv:2307.06783.*

Stoica, O. C. (2023b). Asking physics about physicalism, zombies, and consciousness. *philsci-archive:22459*.

Stoica, O. C. (2023c). The source code for the partition problem experiment https://github.com/CristinelStoica/Partition-all-at-once.

Toffoli, T. (1980). Reversible computing. In *Automata, Languages and Programming: Seventh Colloquium. Noordwijkerhout, the Netherlands July 14–18, 1980*, pages 632–644. Springer.

Turing, A. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(1):230–265.