Program-size versus Time complexity

Slowdown and speed-up phenomena in the micro-cosmos of small Turing machines

Joost J. Joosten[1], Fernando Soler-Toscano[1], and Hector Zenil[2,3]

[1] Grupo de Lógica, Lenguaje e Información
Departamento de Filosofía, Lógica, y Filosofía de la Ciencia
Universidad de Sevilla
{jjoosten,fsoler}@us.es

[2] Laboratoire d'Informatique Fondamentale de Lille
(CNRS), Université de Lille I
[3] Wolfram Research, Inc.
hectorz@wolfram.com

Abstract. The aim of this paper is to undertake an experimental investigation of the trade-offs between program-size and time computational complexity. The investigation includes an exhaustive exploration and systematic study of the functions computed by the set of all 2-color Turing machines with 2 states –we will write (2,2)– and 3 states –we write (3,2)– with particular attention to the runtimes, space-usages and patterns corresponding to the computed functions when the machines have access to larger resources (more states).

We report that the average runtime of Turing machines computing a function almost surely increases as a function of the number of states, indicating that machines not terminating (almost) immediately tend to occupy all the resources at hand. We calculated all time complexity classes to which the algorithms computing the functions found in both (2,2) and (3,2) belong to, and made comparison among these classes. For a selection of functions the comparison is extended to (4,2).

Our study revealed various structures in the micro-cosmos of small Turing Machines. Most notably we observed "phase-transitions" in the halting-probability distribution. Moreover, it is observed that small initial segments fully define a function computed by a TM.

Keywords: small Turing machines, Program-size complexity, Kolmogorov-Chaitin complexity, space/time complexity, computational complexity, algorithmic complexity.

1 Introduction

Among the several measures of computational complexity there are measures focusing on the minimal description of a program and others quantifying the resources (space, time, energy) used by a computation. This paper is a reflection of an ongoing project with the ultimate goal of contributing to the understanding of relationships between various measures of complexity by means of computational experiments. In particular in the current paper we did the following.

We focused on small Turing Machines and looked at the kind of functions that are computable on them focussing on the runtimes. We then study how allowing more computational resources in the form of Turing machine states affect the runtimes of TMs computing these functions. We shall see that in general and on average, more resources leads to slower computations. In this introduction we shall briefly introduce the main concepts central to the paper.

1.1 Two measures of complexity

The long run aim of the project focuses on the relationship between various complexity measures, particularly descriptional and computational complexity measures. In this subsection we shall briefly and informally introduce them.

In the literature there are results known to theoretically link some complexity notions. For example, in [6], runtime probabilities were estimated based on Chaitin's heuristic principle as formulated in [5]. Chaitin's principle is of descriptive theoretic nature and states that *the theorems of a finitely-specified theory cannot be significantly more complex than the theory itself.*

Bennett's concept of logical depth combines the concept of time complexity and program-size complexity [1, 2] by means of the time that a decompression algorithm takes to decompress an object from its shortest description.

Recent work by Neary and Woods [16] has shown that the simulation of cyclic tag systems by cellular automata is effected with a polynomial slow-down, setting a very low threshold of possible non-polynomial tradeoffs between program-size and computational time complexity.

Computational Complexity Computational complexity [4, 11] analyzes the difficulty of computational problems in terms of computational resources. The computational time complexity of a problem is the number of steps that it takes to solve an instance of the problem using the most efficient algorithm, as a function of the size of the representation of this instance.

As widely known, the main open problem with regard to this measure of complexity is the question of whether problems that can be solved in non-deterministic polynomial time can be solved in deterministic polynomial time, aka the P versus NP problem. Since P is a subset of NP, the question is whether NP is contained in P. If it is, the problem may be translated as, for every Turing machine computing an NP function there is (possibly) another Turing machine that does so in P time. In principle one may think that if in a space of all Turing machines with a certain fixed size there is no such a P time machine for the given function (and because a space of smaller Turing machines is always contained in the larger) only by adding more resources a more efficient algorithm, perhaps in P, might be found. We shall see that adding more resources almost certainly yields to slow-down.

Descriptional Complexity The algorithmic or program-size complexity [10, 5] of a binary string is informally defined as the shortest program that can produce the string. There is no algorithmic way of finding the shortest algorithm that outputs a given string

More precisely, the complexity of a bit string s is the length of the string's shortest program in binary on a fixed universal Turing machine. A string is said to be complex or random if its shortest description cannot be much more shorter than the length of the string itself. And it is said to be simple if it can be highly compressed. There are several related variants of algorithmic complexity or algorithmic information.

In terms of Turing machines, if M is a Turing machine which on input i outputs string s, then the concatenated string $\langle M, i\rangle$ is a description of s. The size of a Turing machine in terms of the number of states (s) and colors (k) (aka known as symbols) can be represented by the product $s \cdot k$. Since we are fixing the number of colors to $k = 2$ in our study, we increase the number of states s as a mean for increasing the program-size (descriptional) complexity of the Turing machines in order to study any possible tradeoffs with any of the other complexity measures in question, particularly computational (time) complexity.

1.2 Turing machines

Throughout this project the computational model that we use will be that of Turing machines. Turing machines are well-known models for universal computation. This means, that anything that can be computed at all, can be computed on a Turing machine.

In its simplest form, a Turing machine consists of a two-way infinite tape that is divided in adjacent cells. Each cell can be either blank or contain a non-blank color (symbol). The Turing machine comes with a "head" that can move over the cells of the tape. Moreover, the machine can be in various states. At each step in time, the machine reads what color is under the head, and then, depending on in what state it is writes a (possibly) new color in the cell under the head, goes to a (possibly) new state and have the head move either left or right. A specific Turing machine is completely determined by its behavior at these time steps. One often speaks of a transition rule, or a transition table. Figure 1 depicts graphically such a transition rule when we only allow for 2 colors, black and white and where there are two states, State 1 and State 2.

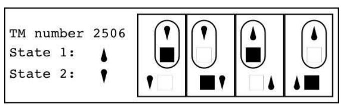

Fig. 1. Transition table of a 2-color 2-state Turing machine with Rule 2506 according to Wolfram's enumeration and Wolfram's visual representation style [14]. [8].

For example, the head of this machine will only move to the right, write a black color and go to State 2 whenever the machine was in State 2 and it read a blank symbol.

We shall often refer to the collection of TMs with k colors and s states as a TM space. From now on, we shall write (2,2) for the space of TMs with 2 states and 2 colors, and (3,2) for the space of TMs with 3 states and 2 colors, etc.

1.3 Relating notions of complexity

We relate and explore throughout the experiment the connections between descriptional complexity and time computational complexity. One way to increase the descriptional complexity of a Turing machine is enlarging its transition table description by adding a new state. So what we will do is, look at time needed to perform certain computational tasks first with only 2 states, and next with 3 and 4 states.

Our current findings suggest that even if a more efficient Turing machine algorithm solving a problem instance may exist, the probability of picking a machine algorithm at random among the TMs that solve the problem in a faster time has probability close to 0 because the number of slower Turing machines computing a function outnumbers the number of possible Turing machines speeding it up by a fast growing function.

1.4 Investigating the micro-cosmos of small Turing machines

We know that small programs are capable of great complexity. For example, computational universality occurs in cellular automata with just 2 colors and nearest neighborhood (Rule 110, see [14, 3]) and also (weak) universality in Turing machines with only 2-states and 3-colors [15].

For all practical purposes one is restricted to perform experiments with small Turing machines (TMs) if one pursuits a thorough investigation of complete spaces for a certain size. Yet the space of these machines is rich and large enough to allow for interesting and insightful comparison, draw some preliminary conclusions and shed light on the relations between measures of complexity.

As mentioned before, in this paper, we look at TMs with 2 states and 2 colors and compare them to TMs more states. The main focus is on the functions they compute and the runtimes for these functions However, along our investigation we shall deviate from time to time from our main focus and marvel at the rich structures present in what we like to refer to as *the micro-cosmos of small Turing machines*. Like, what kind of, and how many functions are computed in each space? What kind of runtimes and space-usage do we typically see and how are they arranged over the TM space? What are the sets that are definable using small Turing machines? How many input values does one need to fully determine the function computed by a TM? We find it amazing how rich the encountered structures are even when we use so few resources.

1.5 Plan of the paper

After having introduced the main concepts of this paper and after having set out the context in this section, the remainder of this paper is organized as follows. In Section 2 we will in full detail describe the experiment, its methodology and the choices that were made leading us to the current methodology. In Section 3 we present the structures that we found in (2,2). The main focus is on runtimes but a lot of other rich structures are exposed there. In Section 4 we do the same for the space (3,2). Section 5 deals with (4,2) but does not disclose any additional structure of that space as we did not exhaustively search this space. Rather we sampled from this space looking for functions we selected from (3,2). In Section 6 we compare the various TM spaces focussing on the runtimes of TMs that compute a particular function.

2 Methodology and description of the experiment

In this section we shall briefly restate the set-up of our experiment to then fill out the details and motivate our choices. We try to be as detailed as possible for a readable paper. For additional information, source code, figures and obtained data can be requested from any of the authors.

2.1 Methodology in short

It is not hard to see that any computation in (2,2) is also present in (3,2). At first, we look at TMs in (2,2) and compare them to TMs in (3,2). In particular we shall study the functions they compute and the time they take to compute in each space.

The way we proceeded is as follows. We ran all the TMs in (2,2) and (3,2) for 1000 steps for the first 21 input values $0, 1, \ldots, 20$. If a TM does not halt by 1000 steps we simply say that it diverges. We saw that certain TMs defined a regular progression of runtimes that needed more than 1000 steps to complete the calculation for larger input values. For these regular progressions we filled out the values manually as described in Subsection 2.7. Thus, we collect all the functions on the domain $[0, 20]$ computed in (2,2) and (3,2) and investigate and compare them in terms of run-time, complexity and space-usage. We selected some interesting functions from (2,2) and (3,2). For these functions we searched by sampling for TMs in (4,2) that compute them so that we could include (4,2) in our comparison.

Clearly, at the outset of this project we needed to decide on at least the following issues:

1. How to represent numbers on a TM?
2. How to decide which function is computed by a particular TM.
3. Decide when a computation is considered finished.

The next subsections will fill out the details of the technical choices made and provide motivations for these choices. Our set-up is reminiscent of and motivated by a similar investigation in Wolfram's book [14], Chapter 12, Section 8.

2.2 Resources

There are $(2sk)^{sk}$ s-state k-color Turing machines. That means 4 096 in (2,2) and 2 985 984 TMs in (3,2). In short, the number of TMs grows exponentially in the amount of resources. Thus, in representing our data and conventions we should be as economical as possible in using our resources so that exhaustive search in the smaller spaces still remains feasible. For example, an additional halting state will immediately increase the search space[4].

2.3 One-sided Turing Machines

In our experiment we have chosen to work with one-sided TMs. That is to say, we work with TMs with a tape that is unlimited to the left but limited to the right-hand side. One sided TMs are a common convention in the literature just perhaps slightly less common than the two sided convention. The following considerations led us to work with one-sided TMs.

- Efficient (that is, non-unary) number representations are place sensitive. That is to say, the interpretation of a digit depends on the position where the digit is in the number. Like in the decimal number 121, the leftmost 1 corresponds to the centenaries, the 2 to the decades and the rightmost 1 to the units. On a one-sided tape which is unlimited to the left, but limited on the right, it is straight-forward how to interpret a tape content that is almost everywhere zero. For example, the tape . . . 00101 could be interpreted as a binary string giving rise to the decimal number 5. For a two-sided infinite tape one can think of ways to come to a number notation, but all seem rather arbitrary.
- With a one-sided tape there is no need for an extra halting state. We say that a computation simply halts whenever the head "drops off" the tape from the right hand side. That is, when the head is on the extremal cell on the right hand side and receives the instruction to moves right. A two-way unbounded tape would require an extra halting state which, in the light of considerations in 2.2 is undesirable.

On the basis of these considerations, and the fact that some work has been done before in the lines of this experiment [14] that also contributed to motivate our own investigation, we decided to fix the TM formalism and choose the one-way tape model.

2.4 Unary input representation

Once we had chosen to work with TMs with a one-way infinite tape, the next choice is how to represent the input values of the function. When working with two colors, there are basically two choices to be made: unary or binary. However, there is a very subtle point if the input is represented in binary. If we choose for

[4] Although in this case not exponentially so, as halting states define no transitions.

a binary representation of the input, the class of functions that can be computed is rather unnatural and very limited.

The main reason is as follows. Suppose that a TM on input x performs some computation. Then the TM will perform the very same computation for any input that is the same as x on all the cells that were visited by the computation. That is, the computation will be the same for an infinitude of other inputs thus limiting the class of functions very severely. We can make this more precise in Theorem 1 below. The theorem shows that coding the input in k-ary where k is the number of alphabet symbols, severely restricts the class of computable functions in (s, k) . For convenience and for the sake of our presentation, we only consider the binary case, that is, $k = 2$.

Definition 1. *A subset of the natural numbers is called a* strip *if it is of the form $\{a + b \cdot n \mid n \in \omega\}$ for certain fixed natural numbers a and b.*

Definition 2. *A* strip *of a function f is simply f restricted to some subdomain $\mathcal{D}$ where $\mathcal{D}$ is a subset of the natural numbers that is a strip.*

Theorem 1 (The Strips Theorem). *Let f be a function that is calculated by a one sided TM. Then f consists of strips of functions of the form $a + x$.*

Proof. Suppose f halts on input i. Let n be the left-most cell visited by the TM on input i. Clearly, changing the input on the left-hand side of this n-th cell will not alter the calculation. In ohter words, the calculation will be the same for all inputs of the form $x \cdot 2^{n+1} + i$. What will be the output for these respective inputs. Well, let $s_i = f(i)$, then the outputs for these infinitely many inputs will consist of this output s_i together with the part of the tape that was unaltered by the computation. Thus, $f(x \cdot 2^{n+1} + i) = x \cdot 2^{n+1} + f(i) = x \cdot 2^{n+1} + i + (f(i) - i)$: behold our strip of the form $a + x$.

We can see the smallest elements that are defining a strip, the i in the proof above, as sort of prime elements. The first calculation on a TM defines a strip. The next calculation on an input not already in that strip defines a new strip and so forth. Thus, the progressively defined strips define new prime elements and the way they do that is quite similar to Eratosthenes' Sieve. For various sieve-generated sets of numbers it is known that they tend to be distributed like the primes ([18]) in that the prime elements will vanish and only occur with probability $\frac{1}{\log(x)}$. If this would hold for the smallest elements of our strips too, in the limit, each element would belong with probability one to some previously defined strip. And each prime element defines some function of the form $x + a_i$. The contribution of the a_i vanishes in the limit so that we end up with the identity function. In this sense, each function calculated by a one-sided TM would calculate the identity in the limit.

The Strips Theorem is bad for two reasons. Firstly, it shows that the class of computable functions is severely restricted. We even doubt that universal function can occur within this class of functions. And, if universal functions do occur, at the cost of how much coding is that the case. In other words, if possible at all, how strong should the coding mechanism be that is needed to represent a computable problem within the strips functions. Secondly, there is the problem of incomparability of functions computed in TM spaces. It is easily seen that Theorem 1 generalizes to more colors k. Still, the strips will compute functions of the form $a+x$, however the strips themselves will be of the form $x \cdot k^{n+1} + i$. Thus, if at some stage the current project were to be extended and one wishes to study the functions that occur in spaces that use a different number of colors, by the Strips Theorem, this intersection is expected to be very small.

The following theorem tells us that the restriction on the class of computable functions when using k-ary input representation has nothing to do with the fact that the computation was done on a single sided TM and the same phenomena occurs in double-sided TMs.

Theorem 2. *Let f be a function that is calculated by a two sided TM. Then f consists of strips of functions of the form $a + 2^l \cdot x$ with $l \in \mathbb{Z}$.*

Proof. As before the proof is based on the observation that altering the input on that part of the tape that was never visited will not influence the calculation. The only thing to be taken into account now is that the output can be shifted to the right (depending on conventions). So that in the end you see that the function, with 2^m units to the right correspond to 2^n units up, hence a tangent of 2^l for some $l \in \mathbb{Z}$.

On the basis of these considerations we decided to represent the input in unary. Moreover, from a theoretical viewpoint it is desirable to have the empty tape input different from the input zero, thus the final choice for our input representation is to represent the number x by $x+1$ consecutive 1's.

The way of representing the input in unary has two serious draw-backs:

1. The input is very homogeneous. Thus, it can be the case that TMs that expose otherwise very rich and interesting behavior, do not do so when the input consists of a consecutive block of 1's.
2. The input is lengthy so that runtimes can grow seriously out of hand. See also our remarks on the cleansing process below.

We mitigate these objections with the following considerations.

1. Still interesting examples are found. And actually a simple informal argument using the Church-Turing thesis shows that universal functions will live in a canonical way among the thus defined functions.
2. The second objection is more practical and more severe. However, as the input representation is so homogeneous, often the runtime sequences exhibit so much regularity that missing values that are too large can be guessed. We shall do so as described in Subsection 2.7.

2.5 Binary output convention

None of the considerations for the input conventions applies to the output convention. Thus, it is wise to adhere to an output convention that reflects as much information about the final tape-configuration as possible. Clearly, by interpreting the output as a binary string, from the output value the output tape configuration can be reconstructed. Hence, our outputs, if interpreted, will be so as binary numbers.

Definition 3 (Tape Identity). *We say that a TM computes the* tape identity *when the tape configuration at the end of a computation is identical to the tape configuration at the start of the computation.*

The output representation can be seen as a simple operation between systems, taking one representation to another. The main issue is, how does one keep the structure of a system when represented in another system, such that, moreover, no additional essential complexity is introduced.

For the tape identity, for example, one may think of representations that, when translated from one to another system, preserve the simplicity of the function. However, a unary input convention and a binary output representation immediately endows the tape identity with an exponential growth rate. In principle this need not be a problem. However, computations that are very close to the tape identity will give rise to number theoretic functions that are seemingly very complex. However, as we shall see, in our current set-up there will be few occasions where we actually do interpret the output as a number other than for representational purposes. In most of the cases the raw tape output will suffice.

2.6 The Halting Problem and Rice's theorem

By the Halting Problem and Rice's theorem we know that it is in general undecidable to know wether a function is computed by a particular TM and whether two TMs define the same function. The latter is the problem of extensionality (do two TMs define the same function?) known to be undecidable by Rice's theorem. It can be the case that for TMs of the size considered in this paper, universality is not yet attained[5], that the Halting Problem is actually decidable in these small spaces and likewise for extensionallity.

As to the Halting Problem, we simply say that if a function does not halt after 1000 steps, it diverges. Theory tells that the error thus obtained actually drops exponentially with the size of the computation bound [6] and we re-affirmed this in our experiments too as is shown in Figure 2. After proceeding this way, we see that certain functions grow rather fast and very regular up to a certain point

[5] Recent work ([17]) has shown some small two-way infinite tape universal TMs. It is known that there is no universal machine in the space of two-way unbounded tape (2,2) Turing machines but there is known at least one weakly universal Turing machine in (2,3)[14] and it may be (although unlikely) the case that a weakly universal Turing machine in (3,2) exists.

where they start to diverge. These obviously needed more than 1000 steps to terminate. We decided to complete these obvious non-genuine divergers manually. This process is referred to as *cleansing* and shall be addressed with more detail in the next subsection.

As to the problem of extensionality, we simply state that two TMs calculate the same function when they compute (after cleansing) the same outputs on the first 21 inputs 0 through 20 with a computation bound of 1000 steps. We found some very interesting observations that support this approach: for the (2,2) space the computable functions are completely determined by their behavior on the first 3 input values 0,1,2. For the $(3,2)$ space the first 8 inputs were found to be sufficient to determine the function entirely.

2.7 Cleansing the data

As mentioned before, the Halting problem is undecidable so one will always err when mechanically setting a cut-off value for our computations. The choice that we made in this paper was as follows. We put the cut-off value at 1000. After doing so, we looked at the functions computed. For those functions that saw an initial segment with a very regular progression of runtimes, for example 16, 32, 64, 128, 256, 512, -1, -1, we decided to fill out the the missing values in a mechanized way. It is clear that, although better than just using a cut-off value, we will still not be getting all functions like this. Moreover, there is a probability that errors are made while filling out the missing values. However we deem the error not too significant, as we have a uniform approach in this process of filling out, that is, we apply the same process for all sequences, either in (2,2) or in (4,2) etc. Moreover, we know from theory ([6]) that most TMs either halt quickly or never halt at all and we affirmed this experimentally in this paper. Thus, whatever error is committed, we know that the effect of it is eventually only marginally. In this subsection we shall describe the way we mechanically filled out the regular progressions that exceeded the computation bound.

We wrote a so-called predictor program that was fed incomplete sequences and was to fill out the missing values. The predictor program is based on the function `FindSequenceFunction`[6] built-in to the computer algebra system *Mathematica*. Basically, it is not essential that we used `FindSequenceFunction` or any other intelligent tool for completing sequences as long as the cleansing method

[6] `FindSequenceFunction` takes a finite sequence of integer values $\{a_1, a_2, \ldots\}$ and retrieves a function that yields the sequence a_n. It works by finding solutions to difference equations represented by the expression `DifferenceRoot` in *Mathematica*. By default, `DifferenceRoot` uses early elements in the list to find candidate functions, then validates the functions by looking at later elements. `DifferenceRoot` is generated by functions such as `Sum`, `RSolve` and `SeriesCoefficient`, also defined in *Mathematica*. `RSolve` can solve linear recurrence equations of any recurring order with constant coefficients. It can also solve many linear equations (up to second recurring order) with non-constant coefficients, as well as many nonlinear equations. For more information we refer to the extensive online *Mathematica* documentation.

for all TM spaces is applied in the same fashion. A thorough study of the cleansing process, its properties, adequacy and limitations is presented in [19]. The predictor pseudo-code is as follows:

1. Start with the finite sequence of integer values (with `-1` values in the places the machine didn't halt for that input index).
2. Take the first n consecutive non-divergent (convergent) values, where $n \geq 4$ (if there is not at least a segment with 4 consecutive non divergent values then it gives up).
3. Call `FindSequenceFunction` with the convergent segment and the first divergent value.
4. Replace the first divergent value with the value calculated by evaluating the function found by `FindSequenceFunction` for that sequence position.
5. If there are no more `-1` values stop otherwise trim the sequence to the next divergent value and go to 1.

This is an example of a (partial) completion: Let's assume one has a sequence (2, 4, 8, 16, -1, 64, -1, 257, -1, -1) with 10 values. The predictor returns: (2, 4, 8, 16, 32, 64, 128, 257, -1, -1) because up to 257 the sequence seemed to be 2^n but from 257 on it was no longer the case, and the predictor was unable to find a sequence fitting the rest.

The prediction function was constrained by 1 second, meaning that the process stops if, after a second of trying, no prediction is made, leaving the non-convergent value untouched. This is an example of a completed Turing machine output sequence. Given (3, 6, 9, 12, -1, 18, 21, -1, 27, -1, 33, -1) it is retrieved completed as (3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36). Notice how the divergent values denoted by -1 are replaced with values completing the sequence with the predictor algorithm based in *Mathematica's* `FindSequenceFunction`.

The prediction vs. the actual outcome For a prediction to be called successful we require that the output, runtime and space usage sequences coincide in every value with the step-by-step computation (after verification). One among three outcomes are possible:

- Both the step-by-step computation and the sequences obtained with `FindSequenceFunction` completion produce the same data, which leads us to conclude that the prediction was accurate.
- The step-by-step computation produces a non-convergent value -1, meaning that after the time bound the step-by-step computation didn't produce any new convergent value that wasn't also produced by the FindSequeceFunction (which means that either the value to be produced requires a larger time bound, or that the `FindSequenceFunction` algorithm has failed, predicting a convergent value where it is actually divergent).
- The step-by-step computation produced a value that the `FindSequenceFunction` algorithm did not predict.

In the end, the predictor indicated what machines we had to run for larger runtimes in order to complete the sequences up to a final time bound of 200 000 steps for a subset of machines that couldnt be fully completed with the predictor program. The number of incorrectly predicted (or left incomplete) in (3,2) was 90 out of a total 3368 sequences completed with the predictor program. In addition to these 45 cases of incorrect completions, we found 108 cases where the actual computation produced new convergent values that the predictor could not predict. The completion process led us to only eight final non-completed cases, all with super fast growing values.

In (4,2) things werent too different. Among the 30 955 functions that were sampled motivated by the functions computed in (3,2) that were found to have also been computed in (4,2) (having in mind a comparison of time complexity classes) only 71 cases could not be completed by the prediction process, or were differently computed by the step-by-step computation. That is only 0.00229 of the sequences, hence in both cases allowing us to make accurate comparisons with low uncertainty in spite of the Halting Problem and the problem of very large (although rare) halting times.

2.8 Running the experiment

To explore the different spaces of TMs we wrote a TM simulator in the programming language C. We tested this C language simulator against the `TuringMachine` function in *Mathematica* as it used the same encoding for TMs. It was checked and found in concordance for the whole (2,2) space and a sample of the (3,2) space.

We run the simulator in the cluster of the CICA (Centro de Informática Científica de Andalucía[7]). To explore the (2,2) space we used only one node of the cluster and it took 25 minutes. The output was a file of 2 MB. For (3,2) we used 25 nodes (50 microprocessors) and took a mean of three hours in each node. All the output files together fill around 900 MB.

3 Investigating the space of 2-states, 2-colors Turing machines

In this section we shall have our first glimpse into the fascinating micro-cosmos of small Turing machines. We shall see what kind of computational behavior is found among the functions that live in (2,2) and reveal various complexity-related properties of the (2,2) space.

Definition 4. *In our context and in the rest of this paper, an* algorithm *computing a function is one particular set of 21 quadruples of the form*

$$\langle \textit{input value}, \textit{output value}, \textit{runtime}, \textit{space usage} \rangle$$

for each of the input values $0, 1, \ldots, 20$*, where the output, runtime and space-usage correspond to that particular input.*

[7] Andalusian Centre for Scientific Computing: `http://www.cica.es/`.

In the cleansed data of (2,2) we found 74 functions and a total of 138 different algorithms computing them.

3.1 Determinant initial segments

An indication of the complexity of the (2,2) space is the number of inputs needed to determine a function. In the case of (2,2) this number of inputs is only 3. For the first input, the input 0, there are 11 different outputs. The following list shows these different outputs (first value in each pair) and the frequency they appear with (second value in each pair). Output `-1` represents the divergent one:

```
{{3, 13}, {2, 12}, {-1, 10}, {0, 10}, {1, 10}, {7, 6}, {6, 4},
 {15, 4}, {4, 2}, {5, 2}, {31, 1}}
```

For two inputs there are 55 different combinations and for three we find all the 74 functions. The first input is most significant; without it, the other inputs only appear in 45 different combinations. This is because there are many functions with different behavior for the first input than for the rest.

We find it interesting that only 3 values of a TM are needed to fully determine its behavior in the full (2,2) space that consists of 4 096 different TMs. Just as a matter of analogy we bring the $\mathbf{C}^\infty$ functions to mind. These infinitely often differentiable continuous functions are fully determined by the outputs on a countable set of input values. It is an interesting question how the minimal number of input values needed to determine a TM grows relative to the total number of $(2 \cdot s \cdot k)^{s \cdot k}$ many different TMs in (s, k) space, or relative to the number of defined functions in that space.

3.2 Halting probability

In the cumulative version of Figure 2 we see that more than 63% of executions stop after 50 steps, and little growth is obtained after more steps. Considering that there is an amount of TMs that never halt, it is consistent with the theoretical result in [6] that most TMs stop quickly or never halt.

Let us briefly comment on Figure 2. First of all, we stress that the halting probability ranges over all pairs of TMs in (2,2) and all inputs between 0 and 20. Second, it is good to realize that the graph is some sort of best fit and leaves out zero values in the following sense. It is easy to see that on the one-sided TM halting can only occur after an odd number of steps. Thus actually, the halting probability of every even number of steps is zero. This is not so reflected in the graph because of a smooth-fit.

We find it interesting that Figure 2 shows features reminiscent of phase transitions. Completely contrary to what we would have expected, these "phase transitions" were even more pronounced in $(3, 2)$ as one can see in Figure 12.

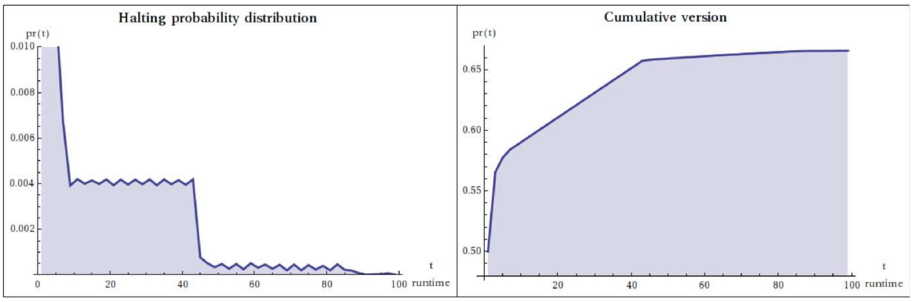

Fig. 2. Halting times in (2,2).

3.3 Phase transitions in the halting probability distribution

Let consider Figure 2 again. Note that in this figure only pairs of TMs and inputs are considered that halt in at most 100 steps. The probability of stopping (a random TM in $(2,2)$ with a random input in 0 to 20) in at most 100 steps is 0.666. The probability of stopping in any number of steps is 0.667, so most TMs stop quickly of do not stop.

We clearly observe a phase transition phenomenon. To investigate the cause of this, let us consider the set of runtimes and the number of their occurrences. Figure 3 shows at the left the 50 smallest runtimes and the number of occurrences in the space that we have explored. The phase-transition is apparently caused

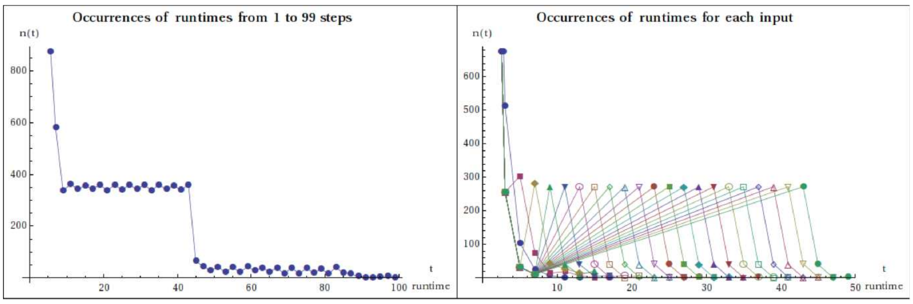

Fig. 3. Occurrences of runtimes

because there are some blocks in the runtimes. To study the cause of this phase-transition we should observe that the left diagram on Figure 3 represents the occurrences of runtimes for arbitrary inputs from 0 to 20. The graph on the right of Figure 3 is clearer. Now, lines correspond to different inputs from 0 to 20. The graph at the left can be obtained from the right one by adding the occurrences corresponding to points with the same runtime. The distribution that we observe here explains the phase-transition effect. It's very interesting that in all cases there is a local maximum with around 300 occurrences and after this maximum, the evolution is very similar. In order to explain this, we look at the following list[8] that represents the 10 most frequent runtime sequences in $(2,2)$. Every runtime sequence is preceded by the number of TMs computing it:

2048	{1, 1, ...}	106	{-1, 3, 3, ...}	20	{3, 7, 11, 15, ...}
1265	{-1, -1, ...}	76	{3, -1, -1, ...}	20	{3, 5, 5, ...}
264	{3, 5, 7, 9, ...}	38	{5, 7, 9, 11, ...}		
112	{3, 3, ...}	32	{5, 3, 3, ...}		

We observe that there are only 5 sequences computed more than 100 times. They represent 92.65% of the TMs in $(2,2)$. There is only one sequence that is not constant nor divergent (recall that -1 represents divergences) with 264 occurrences: $\{3,5,7,9,\ldots\}$. That runtime sequence corresponds to TMs that give a walk forth and back over the input tape to run of the tape and halt. This is the most trivial linear sequence and explains the intermediate step in the phase-transition effect. There is also another similar sequence with 38 occurrences $\{5,7,9,11,\ldots\}$. Moreover, observe that there is a sequence with 20 occurrences where subsequent runtimes differ by 4 steps. This sequence $\{3,7,11,15,\ldots\}$ contains alternating values of our original one $\{3,5,7,9,\ldots\}$ and it explains the zigzag observed in the left part of Figures 2 and 3.

Altogether, this analysis accounts for the observed phase transition. In a sense, the analysis reduces the phase transition to the strong presence of linear performers in Figure 18 together with the facts that on the one hand there are few different kinds of linear performers and on the other hand that each group of similar linear TMs is "spread out over the horizontal axis" in Figure 2 as each input 0,..., 20 is taken into account.

3.4 Runtimes

There is a total of 49 different sequences of runtimes in (2,2). This number is 35 when we only consider total functions. Most of the runtimes grow linear with the size of the input. A couple of them grow quadratically and just two grow exponentially. The longest halting runtime occurs in TM numbers 378 and 1351, that run for 8 388 605 steps on the last input, that is on input 20. Both TMs used only 21 cells[9] for their computation and outputted the value 2 097 151.

Rather than exposing lists of outputvalues we shall prefer to graphically present our data. The sequence of output values is graphically represented as follows. On the top line we depict the tape output on input zero (that is, the input consisted of just one black cell). On the second line immediately below the first one, we depict the tape output on input one (that is, the input consisted of two black cells), etc. By doing so, we see that the function computed by TM 378 is just the tape identity.

[8] The dots denote a linear progression (or constant which is a special case of linear).

[9] It is an interesting question how many times each cell is visited. Is the distribution uniform over the cells? Or centered around the borders?

Let us focus on all the (2,2) TMs that compute that tape identity. We will depict most of the important information in one overview diagram. This diagram as shown in Figure 4 contains at the top a graphical representation of the function computed as described above.

Below the representation of the function, there are six graphs. On each horizontal axis of these graphs, the input is plotted. The τ_i is a diagram that contains plots for all the runtimes of all the different algorithms computing the function in question. Likewise, σ_i depicts all the space-usages occurring. The $<\tau>$ and $<\sigma>$ refer to the (arithmetical) average of time and space usage. The subscript h in e.g. $<\tau>_h$ indicates that the harmonic average is calculated. As the harmonic average is only defined for non-zero numbers, for technical reasons we depict the harmonic average of $\sigma_i + 2$ rather than for σ_i.

Let us recall a definition of the harmonic mean. The harmonic mean of n non-zero values $x_1, \ldots, x_n$ is defined as

$$<x>_h := \frac{n}{\frac{1}{x_1} + \ldots + \frac{1}{x_n}}.$$

In our case, the harmonic mean of the runtimes can be interpreted as follows. Each TM computes the same function. Thus, the total amount of information in the end computed by each TM per input is the same although runtimes may be different. Hence the runtime of one particular TM on one particular input can be interpreted as time/information. We now consider the following situation:

Let the exhaustive list of TMs computing a particular function f be $\{TM_1, \ldots, TM_n$ with runtimes $t_1,$
$\ldots, t_n\}$. If we normalize the amount of information computed by f to 1, we can interpret e.g. $\frac{1}{t_k}$ as the amount of information computed by TM_k in one time step. If we now let TM_1 run for 1 time unit, next TM_2 for 1 time unit and finally TM_n for 1 time unit, then the total amount of information of the output computed is $1/t_1 + ... + 1/t_n$. Clearly,

$$\overbrace{\frac{1}{<\tau>_h} + \ldots + \frac{1}{<\tau>_h}}^{n \text{ times}} = \overbrace{\frac{\frac{1}{t_1} + \ldots + \frac{1}{t_n}}{n} + \ldots \frac{\frac{1}{t_1} + \ldots + \frac{1}{t_n}}{n}}^{n \text{ times}} = \frac{1}{t_1} + \ldots + \frac{1}{t_n}.$$

Thus, we can see the harmonic average as the time by which the typical amount of information is gathered on a random TM that computes f. Alternatively, the harmonic average $<\tau>_h$ is such that $\frac{1}{<\tau>_h}$ is the typical amount of information computed in one time step on a random TM that computes f.

3.5 Clustering in runtimes and space-usages

Observe the two graphics in Figure 5. The left one shows all the runtime sequences in (2,2) and the right one the used-space sequences. Divergences are represented by -1, so they explain the values below the horizontal axis. We find some exponential runtimes and some quadratic ones, but most of them remain linear. All space usages in (2,2) are linear.

The image provides the basic information of the TM outputs depicted by a diagram with each row the output of each of the 21 inputs, followed by the plot figures of the average resources taken to compute the function, preceded by the time and space plot for each of the algorithm computing the function. For example, this info box tells us that there are 1 055 TMs computing the identity function, and that these TMs are distributed over just 12 different algorithms (i.e. TMs that take different space/time resources). Notice that at first glance at the runtimes τ_i, they seem to follow just an exponential sequence while space grows linearly. However, from the other diagrams we learn that actually most TMs run in constant time and space. Note that all TMs that run out of the tape in the first step without changing the cell value (the 25% of the total space) compute this function.

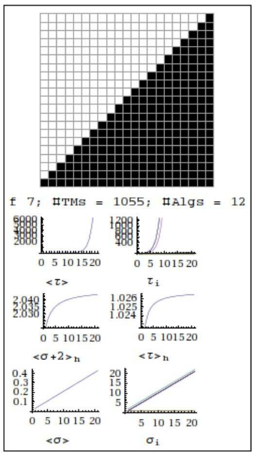

Fig. 4. Overview diagram of the tape identity.

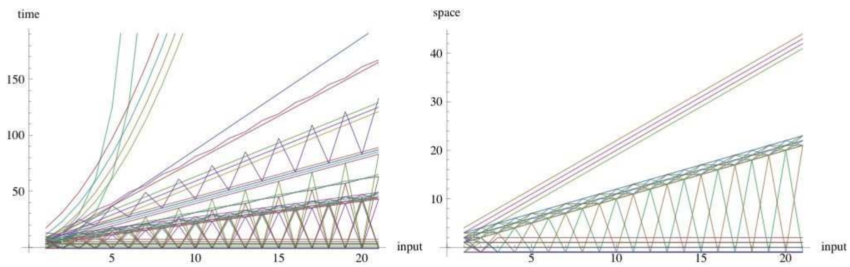

Fig. 5. Runtime and space distribution in (2,2).

An interesting feature of Figure 5 is the clustering. For example, we see that the space usage comes in three different clusters. The clusters are also present in the time graphs. Here the clusters are less prominent as there are more runtimes and the clusters seem to overlap. It is tempting to think of this clustering as rudimentary manifestations of the computational complexity classes.

Another interesting phenomenon is observed in these graphics. It is that of alternating divergence, detected in those cases where value -1 alternates with the other outputs, spaces or runtimes. The phenomena of alternating divergence is also manifest in the study of definable sets.

3.6 Definable sets

Like in classical recursion theory, we say that a set W is definable by a (2,2) TM if there is some machine M such that $W = W_M$ where W_M is defined as usual as

$$W_M := \{x \mid M(x) \downarrow\}.$$

In total, there are 8 definable sets in (2,2). Below follows an enumeration of them.

```
{{}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20}, {0}, {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20},
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20}, {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}, {0, 1}}
```

It is easy to see that the definable sets are closed under complements.

3.7 Clustering per function

We have seen that all runtime sequences in (2,2) come in clusters and likewise for the space usage. It is an interesting observation that this clustering also occurs on the level of single functions. Some examples are reflected in Figure 6.

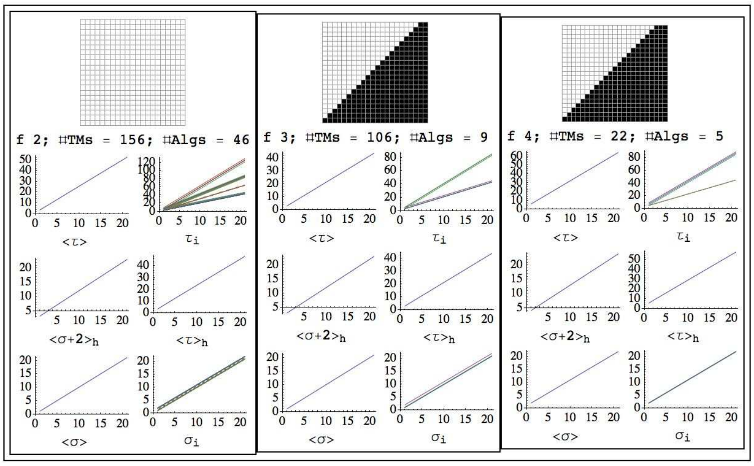

Fig. 6. Clustering of runtimes and space-usage per function.

3.8 Computational figures reflecting the number of available resources

Certain functions clearly reflect the fact that there are only two available states. This is particularly noticeable from the period of alternating converging and non-converging values and in the offset of the growth of the output, and in the alternation period of black and white cells. Some examples are included in Figure 7.

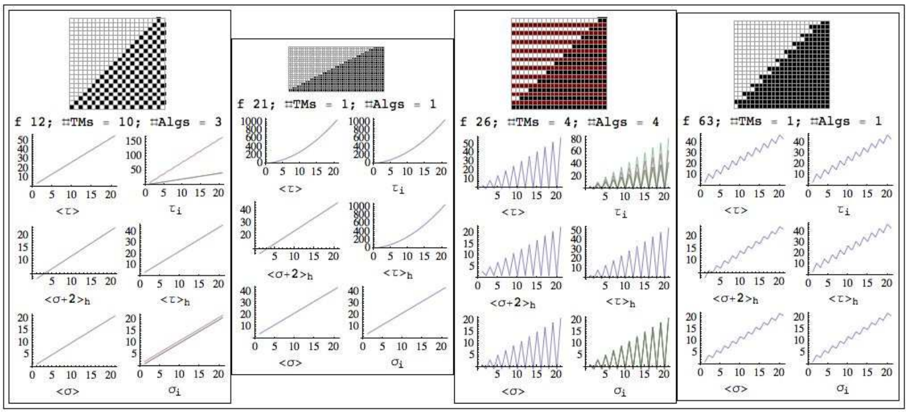

Fig. 7. Computational figures reflecting the number of available resources.

3.9 Types of computations in (2,2)

Let us finish this analysis with some comments about the computations that we can find in (2,2). Most of the TMs perform very simple computations. Apart from the 50% that in every space finishes the computations in just one step (those TMs that move to the right from the initial state), the general pattern is to make just one round through the tape and back. It is the case for TM number 2240 with the sequence of runtimes:

```
{5, 5, 9, 9, 13, 13, 17, 17, 21, 21, ..}
```

Figure 8 shows the sequences of tape configurations for inputs 0 to 5. Each of these five diagrams should be interpreted as follows. The top line represents the tape input and each subsequent line below that represents the tape configuration after one more step in the computation.

The walk around the tape can be more complicated. This is the case for TM number 2205 with the runtime sequence:

```
{3, 7, 17, 27, 37, 47, 57, 67, 77, ...}
```

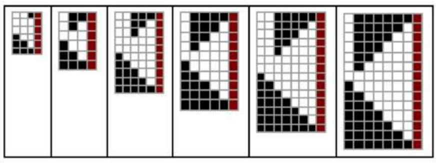

Fig. 8. Turing machine tape evolution for Rule 2240.

which has a greater runtime but it only uses that part of the tape that was given as input, as we can see in the computations (Figure 9, left). TM 2205 is interesting in that it shows a clearly localized and propagating pattern that contains the essential computation.

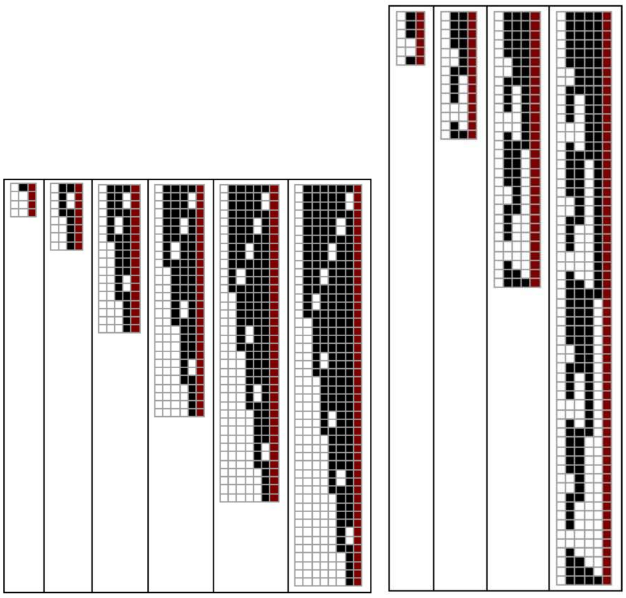

Fig. 9. Tape evolution for Rules 2205 (left) and 1351 (right).

The case of TM 1351 is one of the few that escapes from this simple behavior. As we saw, it has the highest runtimes in (2,2). Figure 9 (right) shows its tape evolution. Note that it is computing the tape identity. Many other TMs in (2,2) compute this function in linear or constant time. In this case of TM 1351 the pattern is generated by a genuine recursive process thus explaining the exponential runtime.

In (2,2) we also witnessed TMs performing iterative computations that gave rise to mainly quadratic runtimes. An example of this is TM 1447, whose computations for the first seven inputs are represented in Figure 10.

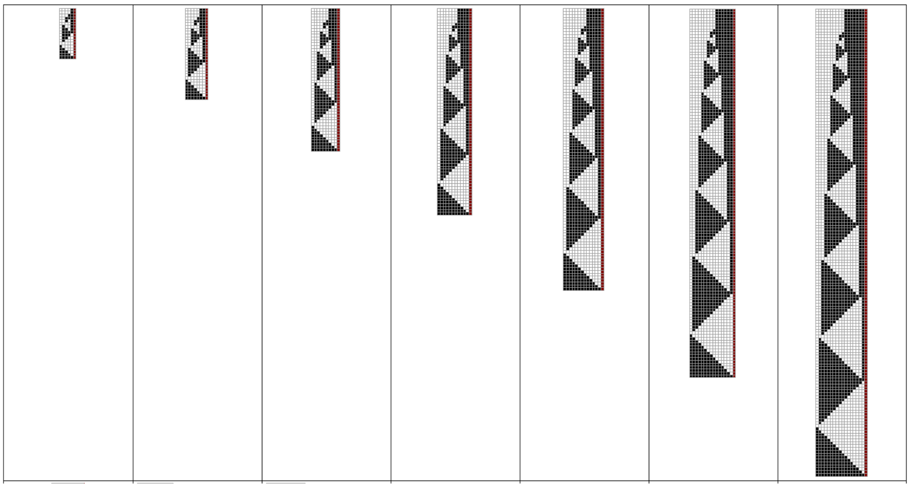

Fig. 10. Turing machine tape evolution for Rule 1447.

Let us briefly summarize the types of computations that we saw in (2,2).

– Constant time behavior like the head (almost) immediately dropping off the tape;
– Linear behavior like running to the end of the tape and then back again as Rule 2240;
– Iterative behavior like using each black cell to repeat a certain process as in Rule 1447;
– Localized computation like in Rule 2205;
– Recursive computations like in Rule 1351.

As most of the TMs in (2,2) compute their functions in the easiest possible way (just one crossing of the tape), no significant speed-up can be expected. Only slowdown is possible in most cases.

4 Investigating the space of 3-states, 2-colors Turing machines

In the cleansed data of (3,2) we found 3886 functions and a total of 12824 different algorithms that computed them.

4.1 Determinant initial segments

As these machines are more complex than those of (2,2), more outputs are needed to characterize a function. From 3 required in (2,2) we need now 8, see Figure 11.

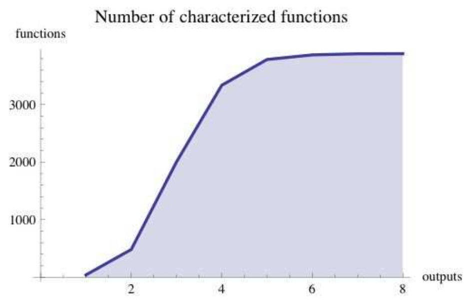

Fig. 11. Number of outputs required to characterize a function in (3,2).

4.2 Halting probability

Figure 12 shows the runtime probability distributions in (3,2). The same behavior that we commented for (2,2) is also observed.

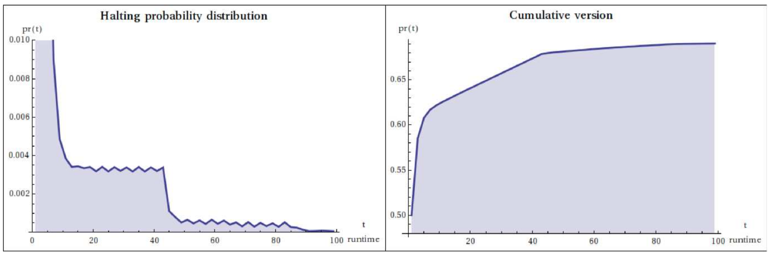

Fig. 12. Runtime proprobability distributions in (3,2).

Note that the "phase transitions" in (3,2) are even more pronounced than in (2,2). We can see these phase transitions as rudimentary manifestations of computational complexity classes. Similar reasoning as in Subsection 3.3 can be applied for (3,2) to account for the phase transitions as we can see in Figure 13.

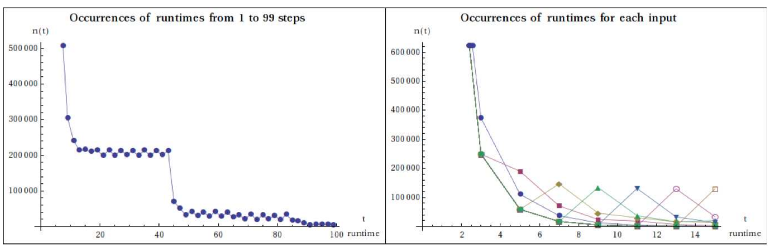

Fig. 13. Occurrences of runtimes

4.3 Runtimes and space-usages

In (3,2) the number of different runtimes and space usage sequences is the same: 3676. Plotting them all as we did for (2,2) would not be too informative in this case. So, Figure 14 shows samples of 50 sequences of space and runtime sequences. Divergent values are omitted as to avoid big sweeps in the graphs caused by the alternating divergers. As in (2,2) we observe the same phenomenon of clustering.

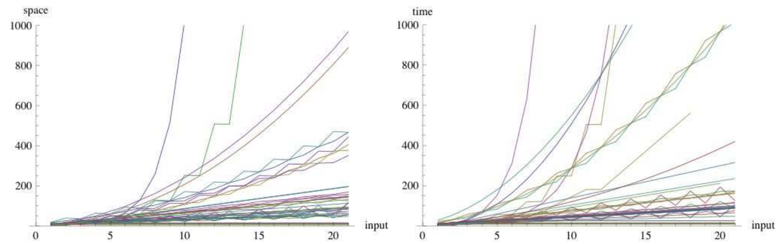

Fig. 14. Sampling of 50 space (left) and runtime (right) sequences in (3,2).

4.4 Definable sets

Now we have found 100 definable sets. Recall that in (2,2) definable sets were closed under taking complements. This does not happen in (3,2). There are 46 definable sets, like

`{{}, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {0, 1, 2}, ...}`

that coexist with their complements, but another 54, like

`{{0, 3}, {1, 3}, {1, 4}, {0, 1, 4}, {0, 2, 3}, {0, 2, 4}, ...}`

are definable sets but their complements are not. We note that, although there are more definable sets in (3,2) in an absolute sense, the number of definable sets in (3,2) relative to the total amount of functions in (3,2) is about four times smaller than in (2,2).

4.5 Clustering per function

In (3,2) the same phenomenon of the clustering of runtime and space usage within a single function also happens. Moreover, as Figure 15 shows, exponential runtime sequences may occur in a (3,2) function (left) while only linear behavior is present among the (2,2) computations of the function (right).

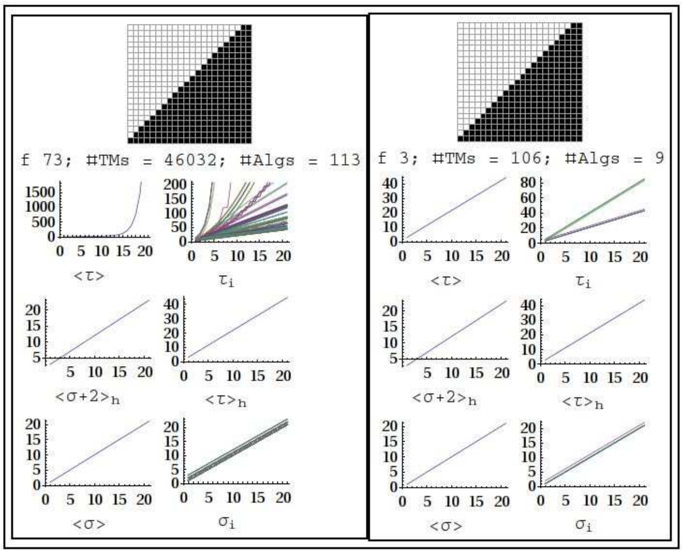

Fig. 15. Clustering per function in (3,2).

4.6 Exponential behavior in (3,2) computations

Recall that in (2,2) most convergent TMs complete their computations in linear time. Now (3,2) presents more interesting exponential behavior, not only in runtime but also in used space.

The max runtime in (3,2) is 894 481 409 steps found in the TMs number 599063 and 666364 (a pair of twin rules[10]) at input 20. The values of this function are double exponential. All of them are a power of 2 minus 2.

Figure 16 shows the tape evolution with inputs 0 and 1. The pattern observed on the right repeats itself.

5 The space (4,2)

An exhaustive search of this space fell out of the scope of the current project. For the sake of our investigations we were merely interested in finding functions in (4,2) that we were interested in. Thus, we sampled and looked only for interesting functions that we selected from (2,2) and (3,2). In searching the 4,2 space, we proceeded as follows. We selected 284 functions in (3,2), 18 of them also in (2,2), that we hoped to find in (4,2) using a sample of about 56x10^6 random TMs.

Our search process consisted of generating random TMs and run them for 1000 steps, with inputs from 0 to 21. The output (with runtime and space usage)

[10] We call two rules in (3,2) *twin rules* whenever they are exactly the same after switching the role of State 2 and State 3.

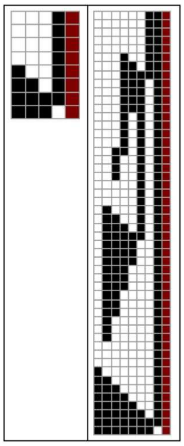

Fig. 16. Tape evolution for Rule 599063.

was saved only for those TMs with a converging part that matches some of the 284 selected functions.

We saved 32235683 TMs. From these, 28032552 were very simple TMs that halt in just one step for every input, so we removed them. We worked with 4203131 non-trivial TMs.

After cleansing there were 1549 functions computed by 49674 algorithms. From these functions, 22 are in (2,2) and 429 in (3,2). TMs computing all the 284 functions of the sampling were found.

Throughout the remainder of the paper it is good to constantly have in mind that the sampling in the (4,2) space is not at all representative.

6 Comparison between the TM spaces

The most prominent conclusion from this section is that when computing a particular function, slow-down of a computation is more likely than speed-up if the TMs have access to more resources to perform their computations. Actually no essential speed-up was witnessed at all. We shall compare the runtimes both numerically and asymptotically.

6.1 Runtimes comparison

In this section we compare the types of runtime progressions we encountered in our experiment. We use the big $\mathcal{O}$ notation to classify the different types of runtimes. Again, it is good to bear in mind that our findings are based on just 21 different inputs. However, the estimates of the asymptotic behavior is based on the functions as found in the cleansing process.

In Figure 17, a table is presented that compares the runtime complexity classes between functions computed in (2,2), (3,2) and (4,2). Under (2,2) is the distribution of time complexity classes for the different algorithms in (2,2) computing the particular function in that row, followed by the distribution of time complexity classes computing the same function in (3,2) and (4,2). Each time complexity class is followed by the number of occurrences among the algorithms in that TM space. The complexity classes are sorted in increasing order.

function #	(2,2)	(3,2)	(4,2)
1	O[1],46 O[n],46	O[1],1109 O[n],1429 O[n^2],7 O[n^3],1	O[1],19298 O[n],28269 O[n^2],77 O[n^3],6
2	O[1],5 O[n],5	O[1],73 O[n],64 O[n^2],7 O[Exp],4	O[1],619 O[n],566 O[n^2],53 O[n^3],16 O[Exp],26
3	O[1],2 O[n],2	O[1],129 O[n],139 O[n^2],2	O[1],2483 O[n],3122 O[n^2],68 O[n^3],1
4	O[1],16 O[n],5 O[Exp],3	O[1],124 O[n],34 O[n^2],9 O[n^3],15 O[n^4],5 O[Exp],15	O[1],1211 O[n],434 O[n^2],101 O[n^3],181 O[n^4],59 O[Exp],156
5	O[1],2 O[n],2	O[1],34 O[n],34	O[1],289 O[n],285 O[n^2],8
6	O[1],3 O[n],3	O[1],68 O[n],74	O[1],576 O[n],668 O[n^2],9 O[n^3],3
7	O[1],10	O[1],54 O[n],8	O[1],368 O[n],94 O[n^3],4 O[Exp],6
8	O[n],1 O[n^2],1	O[n],13 O[n^2],13	O[n],112 O[n^2],107 O[n^3],4 O[Exp],1
9	O[1],2 O[n],2	O[1],58 O[n],54 O[n^2],4	O[1],503 O[n],528 O[n^2],23 O[Exp],4
10	O[n],1 O[n^2],1	O[n],11 O[n^2],11	O[n],114 O[n^2],110 O[n^3],1 O[Exp],3
11	O[n],1 O[n^2],1	O[n],11 O[n^2],11	O[n],91 O[n^2],88 O[n^3],1 O[Exp],2
12	O[n],1 O[n^2],1	O[n],12 O[n^2],12	O[n],120 O[n^2],112 O[n^3],3 O[Exp],5
13	O[1],5 O[n],5	O[1],39 O[n],43	O[1],431 O[n],546 O[n^2],1
14	O[1],4 O[n],4	O[1],14 O[n],14	O[1],119 O[n],121 O[n^2],5 O[n^3],1
15	O[1],2	O[1],11 O[n],1	O[1],69 O[n],15 O[n^2],1 O[Exp],3
16	O[1],18	O[1],27 O[n],7 O[n^3],1 O[Exp],3	O[1],233 O[n],63 O[n^2],15 O[n^3],24 O[n^4],4 O[Exp],29
17	O[1],2 O[n],2	O[1],33 O[n],33	O[1],298 O[n],294 O[n^2],2 O[n^3],2
18	O[1],1 O[n],1	O[1],9 O[n],9	O[1],94 O[n],94
19	O[1],1 O[n],1	O[1],78 O[n],87 O[n^2],1	O[1],1075 O[n],1591 O[n^2],28
20	O[1],1 O[n],1	O[1],15 O[n],15	O[1],76 O[n],75 O[n^2],1
21	O[1],1 O[n],1	O[1],21 O[n],21	O[1],171 O[n],173
22	O[1],1 O[n],1	O[1],14 O[n],14	O[1],203 O[n],203 O[n^2],2 O[Exp],4

Fig. 17. Comparison of the distributions of all 22 functions computed in the three studied spaces (2,2), (3,2) and (4,2). The function number is an index from the list of all the functions computed in these spaces sorted by how they occurred in (2,2).

No essentially (different asymptotic behavior) faster runtime was found in either (3,2) compared to (2,2) or (4,2) compared to (3,2) and (2,2). Thus, no speed-up was found other than by a linear factor as reported in Subsection (6.3). That is, no algorithm in (4,2) or (3,2) computing a function in (3,2) or (2,2) was essentially faster than the fastest algorithm computing already found in (2,2) or (3,2). Amusing findings were Turing machines computing the identify function in as much as exponential time. They are an example of machines spending all resources to compute a simple function. Another example is the constant function f(n) = 0 computed in $O(n^2)$, $O(n^3)$, $O(n^4)$ and even $O(Exp)$.

In (2,2) however, there are very few non-linear time algorithms and functions[11]. However as we see from the similar table for (3,2) versus (4,2) in Figure 17, also between these spaces there is no essential speed-up witnessed. Again only speed-up by a linear factor can occur.

6.2 Distributions over the complexity classes

Figure 18 shows the distribution of the the TMs over the different asymptotic complexity classes. On the level of this distribution we see that the slow-down is manifested in a shift of the distribution to the right of the spectrum.

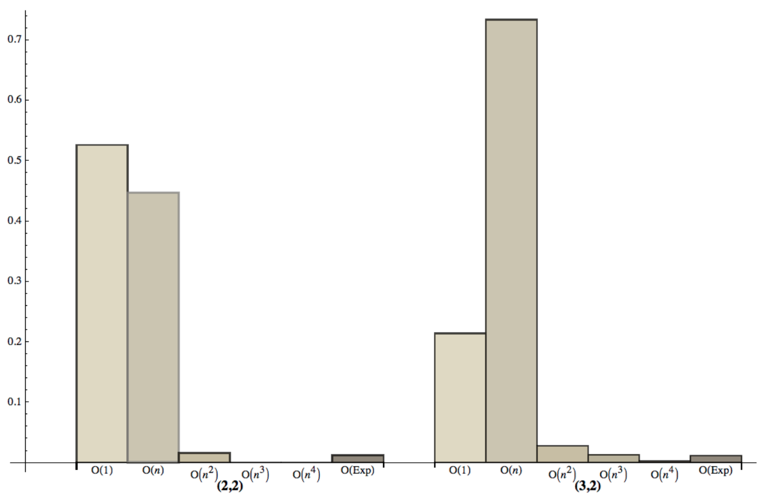

Fig. 18. Time complexity distributions of (2,2) (left) and (3,2) (right).

We have far to few data to possibly speak of a prior in the distributions of our TMs over these complexity classes. However, we do remark the following. In the following table we see the fraction per complexity class of the non-constant TMs for each space. Even though for (4,2) we do not at all work with a representative

[11] We call a function $O(f)$ time, when its asymptotically fastest algorithm is $O(f)$ time.

sampling still there is some similarity in the fractions. Most notably within one TM space, the ratio of one complexity class to another is in the same order of magnitude as the same ratio in one of the other spaces. Notwithstanding this being a far cry from a prior, we do find it worth[12] while mentioning.

pr	(2,2)	(3,2)	(4,2)
O(n)	0.941667	0.932911	0.925167
O(n^2)	0.0333333	0.0346627	0.0462362
O(n^3)	0	0.0160268	0.0137579
O(n^4)	0	0.0022363	0.00309552
O(Exp)	0.025	0.0141633	0.0117433

6.3 Quantifying the linear speed-up factor

For obvious reasons all functions computed in (2,2) are computed in (3,2). The most salient feature in the comparison of the (2,2) and (3,2) spaces is the prominent slowdown indicated by both the arithmetic and the harmonic averages. The space (3,2) spans a larger number of runtime classes. Figures 19 and 20 are examples of two functions computed in both spaces in a side by side comparison with the information of the function computed in (3,2) on the left side and the function computed by (2,2) on the right side. In [9] a full overview of such side by side comparison is published. Notice that the numbering scheme of the functions indicated by the letter f followed by a number may not be the same because they occur in different order in each of the (2,2) and (3,2) spaces but they are presented side by side for comparison with the corresponding function number in each space.

One important calculation experimentally relating descriptional (program-size) complexity and (time resources) computational complexity is the comparison of maximum of the average runtimes on inputs 0,. . .,20, and the estimation of the speed-ups and slowdowns factors found in (3,2) with respect to (2,2).

It turns out that 19 functions out of the 74 computed in (2,2) and (3,2) had at least one fastest computing algorithm in (3,2). That is a fraction of 0.256 of the 74 functions in (2,2). A further inspection reveals that among the 3 414 algorithms in (3,2), computing one of the functions in (2,2), only 122 were faster. If we supposed that “chances” of speed-up versus slow-down on the level of algorithms were fifty-fifty, then the probability that we observed at most 122 instantiations of speed-up would be in the order of 10^{-108}. Thus we can safely state that the phenomena of slow-down at the level of algorithms is significant.

Figure 23 shows the scarceness of the speed-up and the magnitudes of such probabilities. Figures 22 quantify the linear factors of speed-up showing the average and maximum. The typical average speed-up was 1.23 times faster for

[12] Although we have very few data points we could still audaciously calculate the Pearson coefficient correlations between the classes that are inhabited within one of the spaces. Among (2,2), (3,2) and (4,2) the Pearson coefficients are: 0.999737, 0.999897 and 0.999645.

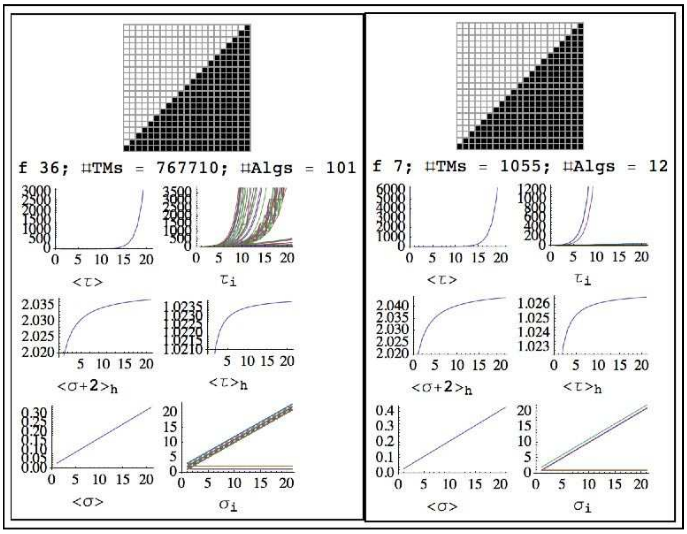

Fig. 19. Side by side comparison of an example computation of a function in (2,2) and (3,2) (the identity function).

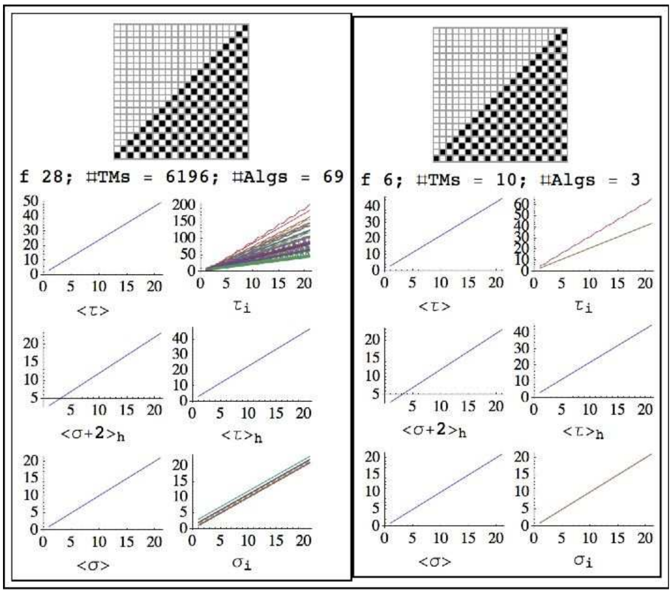

Fig. 20. Side by side comparison of the computation of a function in (2,2) and (3,2).

an algorithm found when there was a faster algorithm in (3,2) computing a function in (2,2).

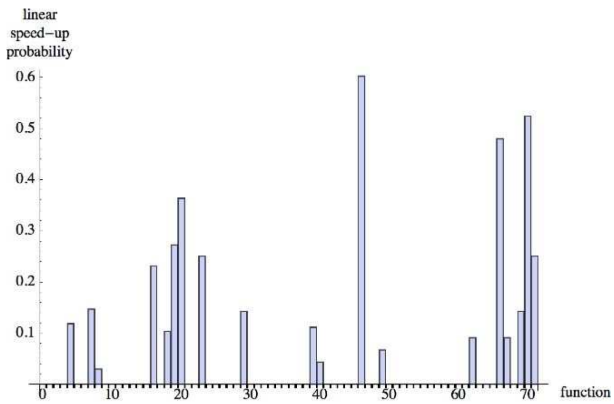

Fig. 21. Distribution of speed-up probabilities per function. Interpreted as the probability of picking an algorithm in (3,2) computing faster an function in (2,2).

In contrast, slowdown was generalized, with no speed-up for 0.743 of the functions. Slowdown was not only the rule but the significance of the slowdown was much larger than the scarce speed-up phenomenon. The average algorithm in (3,2) took 2 379.75 longer and the maximum slowdown was of the order of 1.19837×10^6 times slower than the slowest algorithm computing the same function in (2,2).

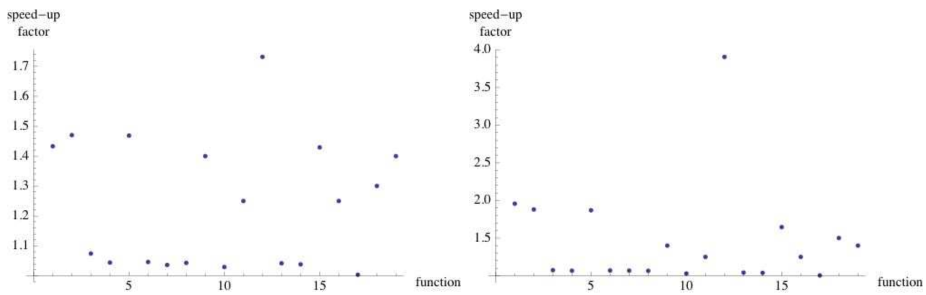

Fig. 22. Speed up significance: on the left average and on the right maximum speed-ups.

As mentioned before there is also no essential speed-up in the space (4,2) compared to (3,2) and only linear speed-up was witnessed at times. But again, slow-down was the rule. Thus, (4,2) confirmed the trend between (2,2) and (3,2), that is that linear speed up is scarce yet present, three functions (0.0069) sampled from (3,2) had faster algorithms in (4,2) that in average took from 2.5 to 3 times less time to compute the same function, see Figure 6.3.

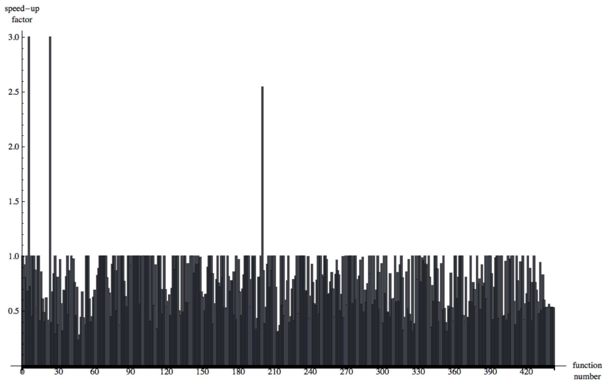

Fig. 23. Distribution of average speed-up factors among all selected 429 functions computed in (3,2) and (4,2).

Acknowledgements

The initial motivation and first approximation of this project was developed during the NKS Summer School 2009 held at the Istituto di Scienza e Tecnologie dell'Informazione, CNR in Pisa, Italy. We wish to thank Stephen Wolfram for interesting suggestions and guiding questions. Furthermore, we wish to thank the CICA center and its staff for providing access to their supercomputing resources and their excellent support.

References

1. C.H. Bennett. Logical Depth and Physical Complexity in Rolf Herken (ed), *The Universal Turing Machine–a Half-Century Survey*; Oxford University Press, p 227-257, 1988.
2. C.H. Bennett. How to define complexity in physics and why, in *Complexity, entropy and the physics of information*, Zurek, W. H.; Addison-Wesley, Eds.; SFI studies in the sciences of complexity, p 137-148, 1990.
3. M. Cook. Universality in Elementary Cellular Automata, *Complex Systems*, 2004.
4. S. Cook. The complexity of theorem proving procedures, *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, p 151-158, 1971.
5. G.J. Chaitin. *Gödel's theorem and information, Int. J. Theoret. Phys.*, 21, p 941-954, 1982.

6. C.S. Calude, M.A. Stay, Most programs stop quickly or never halt, *Advances in Applied Mathematics*, 40, p 295-308, 2005.
7. E. Fredkin. Digital Mechanics, *Physica D*, p 254-70, 1990.
8. J. J. Joosten. Turing Machine Enumeration: NKS versus Lexicographical, *Wolfram Demonstrations Project*; http://demonstrations.wolfram.com/ TuringMachineEnumerationNKSVersusLexicographical/, 2010.
9. J. J. Joosten, F. Soler Toscano, H. Zenil. Turing machine runtimes per number of states, to appear in *Wolfram Demonstrations Project*, 2011.
10. A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information and Transmission*, 1(1), p 1–7, 1965.
11. L. Levin. Universal search problems, *Problems of Information Transmission* 9 (3), p 265-266, 1973.
12. S. Lin & T. Rado. Computer Studies of Turing Machine Problems, *J. ACM*, 12, p 196-212, 1965.
13. S. Lloyd, Programming the Universe; *Random House*, 2006.
14. S. Wolfram, A New Kind of Science; *Wolfram Media*, 2002.
15. Wolfram's 2, 3 Turing Machine Research Prize, http://www.wolframscience.com/prizes/tm23/; Accessed on June, 24, 2010.
16. R. Neary, D. Woods. On the time complexity of 2-tag systems and small universal turing machines, In *FOCS*; IEEE Computer Society, p 439-448, 2006.
17. D. Woods, T. Neary. Small semi-weakly universal Turing machines. *Fundamenta Informaticae*, 91, p 161-177, 2009.
18. M. C. Wunderlich, A general class of sieve generated sequences, *Acta Arithmetica* XVI, p 41-56, 1969.
19. H. Zenil, F. Soler Toscano, J. J. Joosten. Empirical encounters with computational irreducibility and unpredictability, submitted to *International Journal of Foundations of Computer Science*, December 2010.