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 proceeds by an exhaustive exploration and systematic study of the functions computed by the set of all 2-color Turing machines with 2, and 3 states 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.
Our study revealed various structures in the micro-cosmos of small Turing Machines. Most notably we observed "phase-transitions" in the halting-probability distribution.
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.

1.1 Two measures of complexity

The long run aim of the project focuses on the relationship between 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 also 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 [14] 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, 9] 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 solving machine for the given problem (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.

Descriptional Complexity The algorithmic or program-size complexity [8, 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

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) is determined 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 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 a different state. 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.

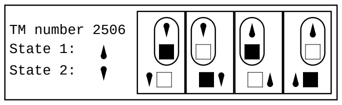


Fig. 1. Transition table of a 2-color 2-state Turing machine with rule 2506 according to Wolfram's enumeration and Wolfram's visual representation style [12].

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.

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. 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 solving 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.

This suggests that the theoretical problem of P versus NP might be disconnected to the question in practice when using brute force techniques. Disregarding the answer to the P versus NP as a theoretical problem, efficient heuristics to search for the P time algorithm may be required, other than picking it at random or searching it by exhaustive means, for otherwise the question in practice may have a different answer in the negative independent of the theoretical solution. We think our approach provides insights in this regard.

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) [12, 3] and also (weak) universality in Turing machines with only 2-states and 3-colors [13].

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.

To be more concrete, in this paper, we look at TMs with 2 states and 2 colors and compare them to TMs with 3 states and 2 colors. The main focus is on the functions they compute and the runtimes for these functions[4]. Some of the questions we try to answer include 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?

2 Methodology

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. Let us briefly restate the set-up of our experiment.

2.1 Methodology in short

We look at TMs in (2,2) and compare them to TMs in (3,2). In particular we shall study the functions they compute[5] and the time they take to compute in each space.

[4] We shall often refer to the collection of TMs with k colors and s states as a TM space.

[5] It is not hard to see that any function that is computed in (2,2) is also present in (3,2).

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. 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.

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 surely motivated by a similar investigation in Stephan Wolfram's book [12], 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 spaces still remains feasible. For example, an additional halting state will immediately increase the search space[6].

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 after the more common 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 $\ldots 00101$ 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.

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

- 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 [12] 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 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. Thus, it will be unlikely that some universal function can be computed for any natural notion of universality.

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 this way 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.

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.

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 complexity is introduced.

For the tape identity (see Definition 2), for example, one may think of representations that, when translated from one to another system, preserve the simplicity of the function. Some will do so such as taking the output in unary. If one uses a unary representation to feed the *Mathematica* function FindSequenceFunction[7] that will find out, by looking at the sequence of outputs in the chosen representation, that it is about the identity function as one would immediately tell upon looking at the pictogram of the Turing machine. But unary does not work for all other Turing machine evolutions.

For example, when taking the output tape configuration as written in binary, many functions expose (at least) exponential growth. For the tape-identity, that is a TM that outputs the same tape configuration as the input tape configuration, the function $TM(x) = 2^{x+1} - 1$ is the sequence generator under this output representation, rather than $TM(x) = x$. In particular, the TM that halts immediately by running off the tape while leaving the first cell black also computes the function $2^{x+1} - 1$.

These concerns, although legitimate and rich in discussion are undesirable, but 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.

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[8], that the halting problem is actually decidable in these small spaces and likewise for extensionallity.

[7] This function in *Mathematica* may be seen as a specific purpose Turing machine for which a compiler is needed so that one can provide as input to this function the output of one of our Turing machines. FindSequenceFunction will then attempt to find a simple function that yields the sequence when given successive integer arguments.

[8] Recent work by [15] have 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 weak universal Turing machine in (2,3)[12] and it may be (although unlikely) the case that a weak universal Turing machine in (3,2) exists.

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 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*, Of course some checks were performed as to give more grounds for doing so. We are fully aware that errors can have occurred in the cleansing. For example, a progression of a TM is guessed and checked for two values. However, it can be the case that for the third value our guess was wrong: the Halting Problem is undecidable and our approximation is better than doing nothing.

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 Running the experiment

To explore the different spaces of TMs we have programmed in C language a TM simulator. 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 have run the simulator in the cluster of the CICA (Centro de Informática Científica de Andalucía[9]). 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 Results

Definition 1. *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*

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

where the output, runtime and space-usage correspond to that particular input.

Definition 2. *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.*

[9] Andalusian Centre for Scientific Computing.

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

In the cleansed data of (2,2) we found 74 functions and a total of 253 different algorithms computing them.

Determinant initial segments An indication of the complexity of the (2,2) space is the number of outputs needed to determine a function. In the case of (2,2) this number of outputs is only 3. For the first output 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 outputs there are 55 different combinations and for three we find the full 74 functions. The first output is most significant; without it, the other outputs 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 output 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.

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.

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 10.

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.

Below follows the sequence of {`input, output, runtime, space`} for TM number 378:

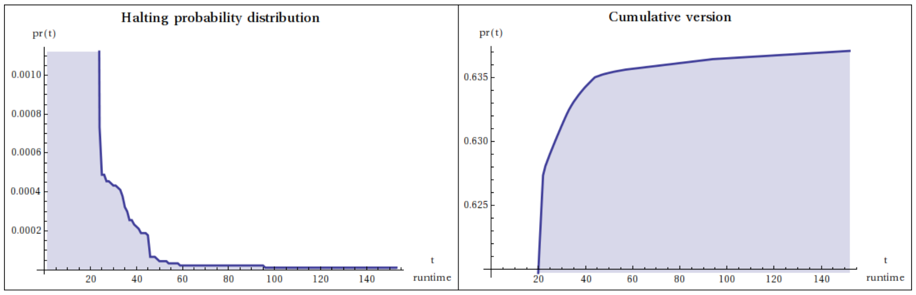


Fig. 2. Halting times in (2,2).

```
{{0, 1, 5, 1}, {1, 3, 13, 2}, {2, 7, 29, 3}, {3, 15, 61, 4},
{4, 31, 125, 5}, {5, 63, 253, 6}, {6, 127, 509, 7}, {7, 255,
1021, 8}, {8, 511, 2045, 9}, {9, 1023, 4093, 10}, {10, 2047,
8189, 11}, {11, 4095, 16381, 12}, {12, 8191, 32765, 13},
{13, 16383, 65533, 14}, {14, 32767, 131069, 15}, {15, 65535,
262141, 16}, {16, 131071, 524285, 17}, {17, 262143, 1048573,
18}, {18, 524287, 2097149, 19}, {19, 1048575, 4194301, 20},
{20, 2097151, 8388605, 21}}
```

Rather than exposing lists of values we shall prefer to graphically present our data. The output values are graphically represented as follows. On the fist line we depict the tape output on input zero (that is, the input consisted of just one black cell). On the second line 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 378 is just the tape identity.

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 3 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 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.

The harmonic mean of the runtimes can be interpreted as follows. Each TM computes the same function. Thus, the total information in the end computed by each TM per entry 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. If we consider the following situation:

Let the TMs computing a function be $\{TM_1, \ldots, TM_n$ with runtimes $t_1, \ldots, t_n\}$.

If we 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 amount of information of the output computed is $1/t_1 + ... + 1/t_n$. The corresponding average of this impact function is exactly the harmonic mean, hence the introduction of the harmonic mean as an interpretation of the typical amount of information computed by a random TM in a time unit.

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.

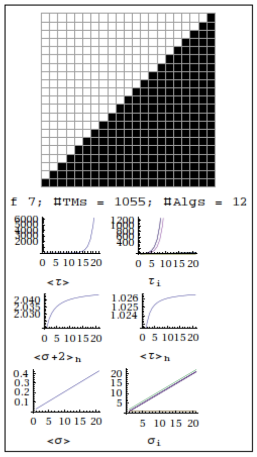


Fig. 3. Overview diagram of the tape identity.

Runtimes and space-usages Observe the two graphics in Figure 4. 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 but most of them and space-usage remain linear.

An interesting feature of Figure 4 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

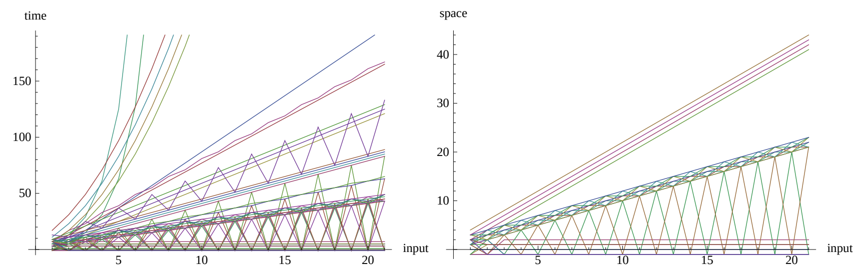


Fig. 4. Runtime and space distribution in (2,2).

the other outputs, spaces or runtimes. The phenomena of alternating divergence is also manifest in the study of definable sets.

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

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

Below follows an enumeration of the definable sets in (2,2).

```
{{}, {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.

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 5.

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 6.

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 finish the computations

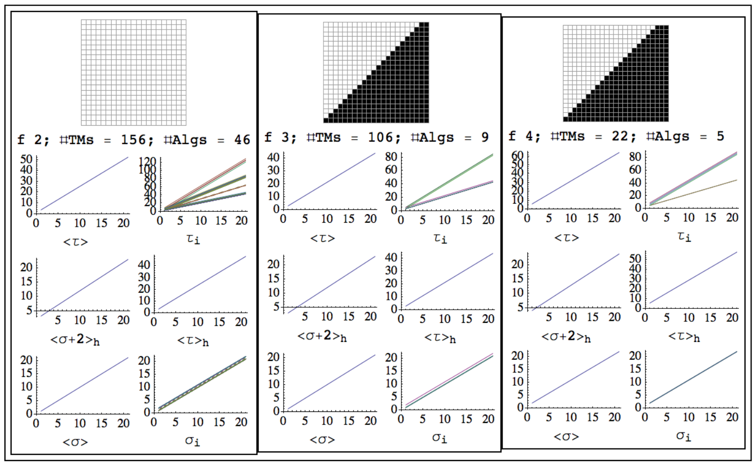


Fig. 5. Clustering of runtimes and space-usage per function.

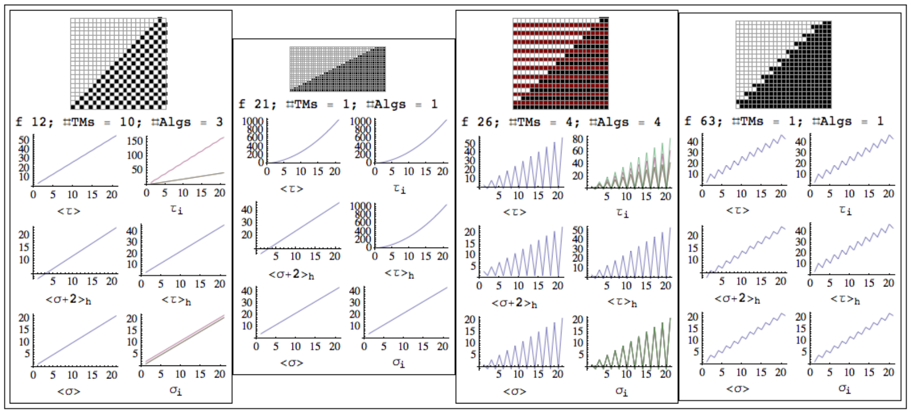


Fig. 6. Computational figures reflecting the number of available resources.

in just one step (those 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, ..}
```

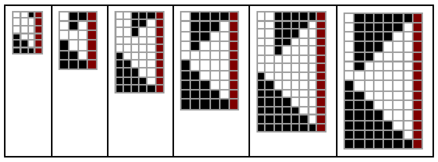


Fig. 7. Turing machine tape evolution for rule 2240.

TM 2205 however is interesting in that it shows a clearly localized and propagating pattern that contains the essential computation. Most TMs that cross the tape just once and then go back to the beginning of the tape expose behavior that is a lot simpler and only visit each cell twice.

Figure 7 shows the sequences of tape configurations for inputs 0 to 5. 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, ...}
```

it 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 8, left). In this case the pattern is generated by a genuine recursive process thus explaining the exponential runtime.

The case of TM 1351 is one of the few that escapes from this simple behavior. As we saw, it has the greatest runtimes in (2,2). Figure 8 (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 (2,2) we also witnessed TMs performing iterative computations that gave rise to mainly quadratic runtimes.

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.

3.2 Investigating the space of 3-state, 2-color Turing machines

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

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 9.

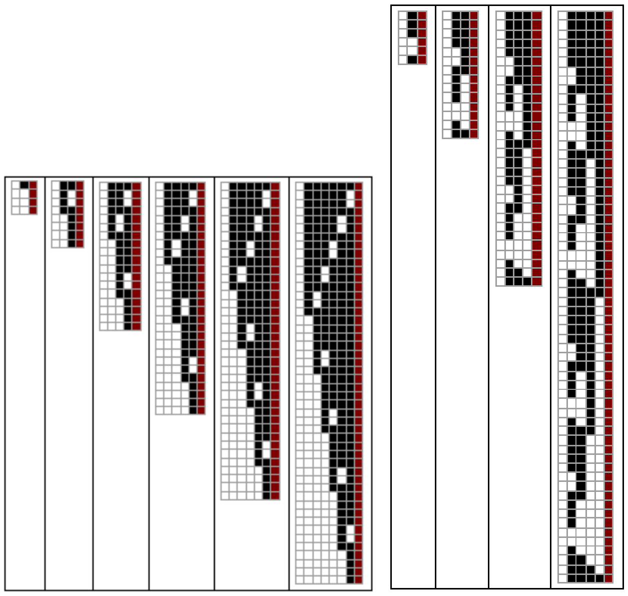


Fig. 8. Tape evolution for rules 2205 (left) and 1351 (right).

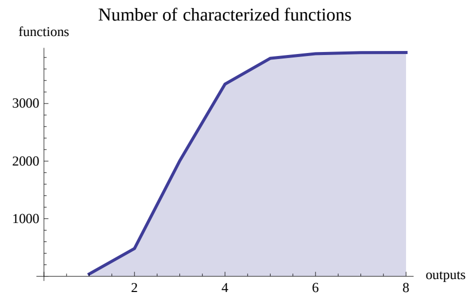


Fig. 9. Number of outputs required to characterize a function in (3,2).

Halting probability Figure 10 shows the runtime probability distributions in (3,2). The same behavior that we commented for (2,2) is also observed. Note that the "phase transitions" in (3,2) are even more pronounced than in (2,2). It is tempting to think as those phase transitions as rudimentary manifestations of computational complexity classes. Further investigation should show whether the distinct regions correspond to particular methods employed by the TMs in that region. Amongst those method we see as most prominent modes of computing the following: running off the tape (almost) immediately; going from the initial head position to the end of the input and back to the beginning again; iterating a particular process several times; recursion.

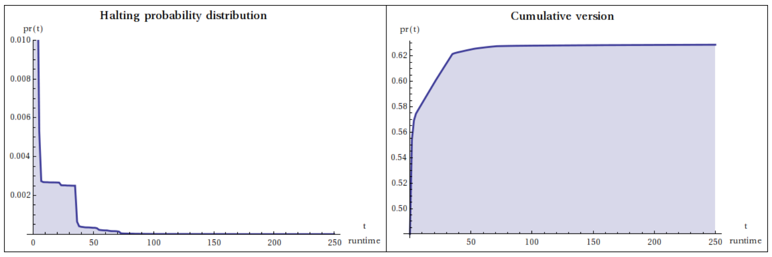


Fig. 10. Runtime proprobability distributions in (3,2).

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 11 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.

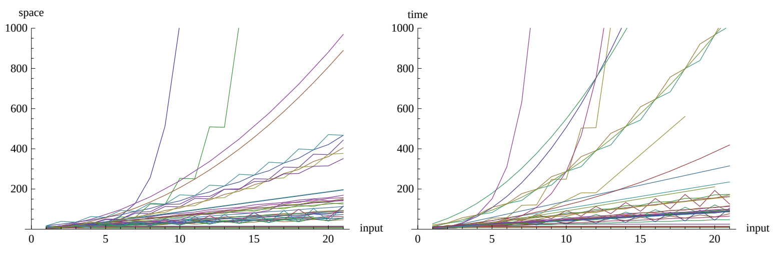


Fig. 11. Sampling of 50 space (left) and runtime (right) sequences in (3,2).

Definable sets Now we have found 100 definable sets. Recall that in (2,2) definable sets were closed under taking complements. It does not happens now. There are 46 definable sets, as

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

that coexist with their complements, but another 54, as

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

are definable sets but their complements are not.

Clustering per function In (3,2) the same phenomenon of the clustering of runtime and space usage in single functions also happens. Moreover, as Figure 12 shows, exponential runtime sequences may occur in a (3,2) function (left) with

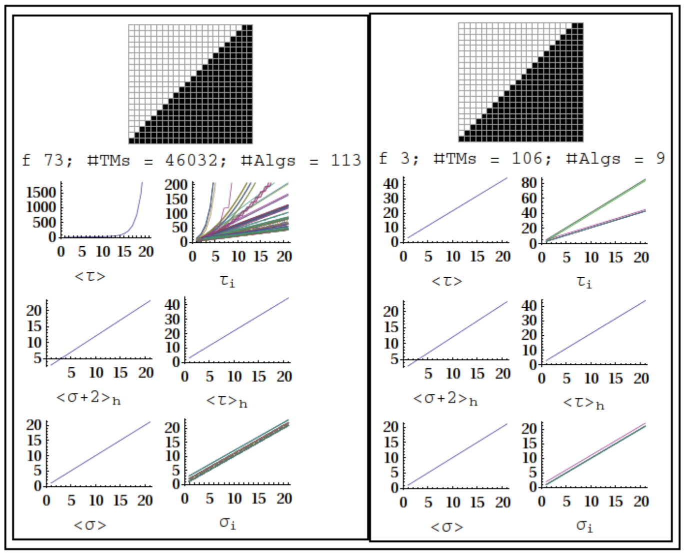


Fig. 12. Clustering per function in (3,2).

other linear behaviors, some of them already present in the (2,2) computations of the function (right).

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. Look at the first outputs:

```
{14, 254, 16382, 8388606, 137438953470, ... }
```

Adding 2 to each value, the logarithm to base 2 of the output sequence is:

```
{4, 8, 14, 23, 37, 58, 89, 136, 206, 311, 469, 706, 1061,
 1594, 2393, 3592, 5390, 8087, 12133, 18202, 27305}
```

Figure 13 displays these logarithms, and the runtime and space sequences.

Finally, Figure 14 shows the tape evolution with inputs 0 and 1. The pattern observed on the right repeats itself.

[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.

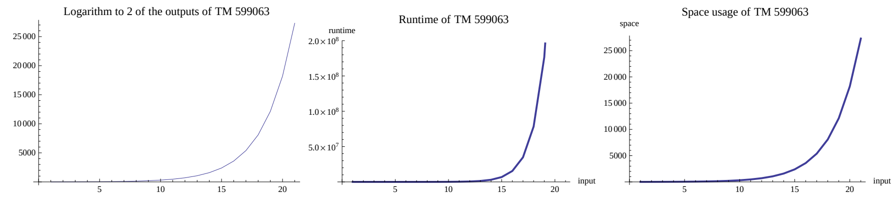


Fig. 13. Rule number 599063. Logarithm to base 2 of the outputs (left), runtime (center) and space usage (right).

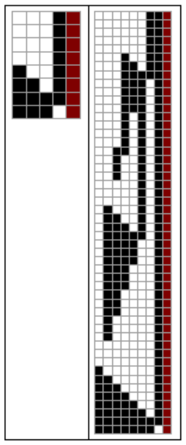


Fig. 14. Tape evolution for rule 599063.

4 Comparison between (2,2) and (3,2)

The most prominent conclusion from this section is that slow-down of a computation is more likely than speed-up.

4.1 Runtimes comparison

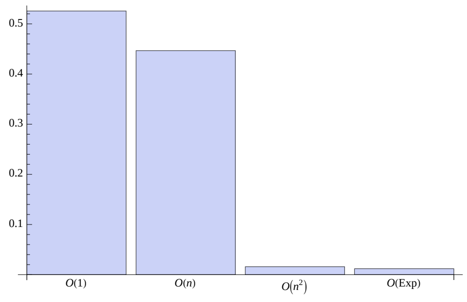
In this section we compare the types of runtime progressions we encountered in our experiment. We use the big $\mathcal{O}$ notation to classify the different types of runtimes. Again, it is clear to bear in mind that our findings are based on just 21 different inputs.

As shown no essentially (different asymptotic behavior) faster runtime was found in (3,2), no speed up was found other than by a linear factor as reported in the next section (4.2). That is, no algorithm in (3,2) computing a function in (2,2) was faster than the fastest algorithm computing the same function in (2,2). Obviously (3,2) computes a larger set of functions and they shall be compared to the next larger (4,2) space of TMs. Amusing findings were Turing machines both in (2,2) and (3,2) computing the identify function in as much as exponential time, as an example of machines spending all resources to compute a simple function. Another example is the constant function $f(n) = 0$ computed in n^9 or n^19, and $f(n) = 1$ computed in as much as exponential time as well, these in (3,2).

In the table, the first column is the function index from 1 to 74 occurred in both (2,2) and (3,2). Under (2,2) is the distribution of time complexity classes for the function in that row in (2,2), followed by the distribution of time complexity classes computing the same function in (3,2). Each time complexity class is followed by the number of occurrences among the algorithms in that TM space and for each function, sorted from greater to lower. No complexity class is estimated if the sequence is divergent, such as for function 1.

Function #	$(\mathbf{2}, \mathbf{2})$	$(\mathbf{3}, \mathbf{2})$
1	*None*	*None*
2	$O(n), 46$	$O(n), 1084$; $O(1), 129$; $O(n^{19}), 46$ $O(n^3), 8$; $O(n^2), 6$
3	$O(n), 9$	$O(n), 93$; $O(n^2), 12$; $O(Exp), 5$ $O(1), 2$; $O(n^{19}), 1$
4	$O(n), 5$	$O(n), 60$; $O(n^2), 9$; $O(Exp), 4$ $O(n^{19}), 1$
5	$O(n), 2$	$O(n), 133$; $O(n^2), 2$
6	$O(n), 3$	$O(n), 61$; $O(1), 7$; $O(n^3), 1$
7	$O(n), 5$; $O(1), 4$; $O(Exp), 3$	$O(1), 46$; $O(n), 32$; $O(Exp), 17$ $O(n^2), 6$
8	$O(n), 2$	$O(n), 34$
9	$O(n), 1$	$O(n), 34$
10	$O(n), 1$	$O(n), 12$; $O(n^2), 1$
11	$O(n), 2$	$O(n), 25$; $O(n^2), 4$; $O(Exp), 2$
12	$O(n), 3$	$O(n), 70$; $O(n^2), 1$
13	$O(1), 2$	$O(1), 12$
14	$O(1), 5$	$O(1), 23$; $O(n), 8$
15	$O(1), 3$	$O(1), 11$
16	$O(1), 3$	$O(1), 9$
17	$O(n^2), 1$	$O(n^2), 13$
18	$O(n), 1$	$O(n), 12$
19	$O(n), 2$	$O(n), 54$; $O(n^2), 4$
20	$O(n^2), 1$	$O(n^2), 11$
21	$O(n^2), 1$	$O(n^2), 11$
22	$O(n), 1$	$O(n), 14$
23	$O(1), 3$	$O(1), 9$
24	$O(n^2), 1$	$O(n^2), 12$
25	$O(n), 5$	$O(n), 38$; $O(n^9), 2$; $O(n^2), 1$
26	$O(n), 4$	$O(n), 14$
27	$O(1), 1$	$O(1), 6$
28	$O(1), 1$	$O(1), 7$

Function #	$(\mathbf{2},\mathbf{2})$	$(\mathbf{3},\mathbf{2})$
29	$O(1), 39$	$O(1), 107$
30	$O(1), 1$	$O(1), 7$
31	$O(1), 3$	$O(1), 25$
32	$O(1), 1$	$O(1), 5; O(n), 1$
33	$O(1), 9$	$O(1), 9; O(n), 7; O(Exp), 3$
34	$O(1), 23$	$O(1), 58; O(n), 13; O(Exp), 1$
35	$O(n), 2$	$O(n), 31; O(n^2), 2$
36	$O(n), 1$	$O(n), 19; O(1), 3$
37	$O(n), 1$	$O(n), 12$
38	$O(1), 1$	$O(1), 23; O(n), 1$
39	$O(1), 1$	$O(1), 16$
40	$O(n), 1$	$O(n), 6; O(1), 3$
41	$O(1), 1$	$O(1), 23$
42	$O(1), 4$	$O(1), 42; O(n), 1$
43	$O(1), 2$	$O(1), 16$
44	$O(1), 1$	$O(1), 22; O(n), 1$
45	$O(1), 1$	$O(1), 8$
46	$O(1), 1$	$O(1), 14; O(n), 2$
47	$O(n), 1$	$O(n), 57; O(1), 26$
48	$O(n), 1$	$O(n), 32$
49	$O(n), 1$	$O(n), 17; O(1), 14$
50	$O(n), 1$	$O(n), 15$
51	$O(n), 1$	$O(n), 15$
52	$O(n), 1$	$O(n), 12$
53	$O(1), 1$	$O(1), 10$
54	$O(1), 3$	$O(1), 70$
55	$O(1), 3$	$O(1), 17; O(n), 1$
56	$O(1), 6$	$O(1), 35; O(n), 7$
57	$O(1), 1$	$O(1), 21; O(n), 4; O(Exp), 2$
58	$O(1), 1$	$O(1), 22$
59	$O(1), 1$	$O(1), 15; O(n), 7$
60	$O(n), 1$	$O(n), 37; O(1), 1$
61	$O(n), 1$	$O(n), 45; O(1), 3$
62	$O(n), 1$	$O(n), 20; O(1), 15$
63	$O(n), 1$	$O(n), 11$
64	$O(n), 1$	$O(n), 31; O(1), 2$
65	$O(n), 1$	$O(n), 21$
66	$O(1), 1$	$O(1), 20$
67	$O(1), 1$	$O(1), 25$
68	$O(1), 1$	$O(1), 11$
69	$O(1), 2$	$O(1), 16$
70	$O(n), 1$	$O(n), 4; O(1), 3$
71	$O(n), 1$	$O(n), 20; O(1), 1$
72	$O(1), 1$	$O(1), 4$
73	$O(n), 1$	$O(n), 10$
74	$O(n), 1$	$O(n), 12; O(1), 2$



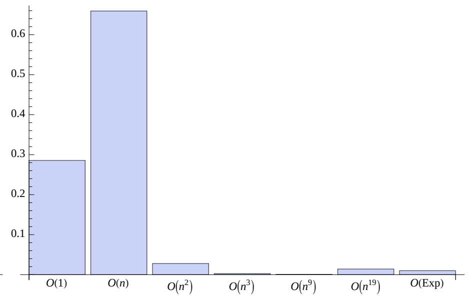


Fig. 15. Time complexity distributions of (2,2) (left) and (3,2) (right).

As shown in this time complexity table comparing runtimes between (2,2) and (3,2), no speed up was found other than by a linear factor as reported in the next subsection (4.2). That is, no algorithm in (3,2) computing a function in (2,2) was faster than the fastest algorithm computing the same function in (2,2). Obviously (3,2) computes a larger set of functions and they shall be compared to the next larger (4,2) space of TMs. An amusing finding were Turing machines both in (2,2) and (3,2) computing the identify function in as much as exponential time, as an example of a machine spending all resources to compute a simple function.

4.2 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. (3,2) spans a larger number of runtime classes. Figures 16 and 17 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. 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 averages 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 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

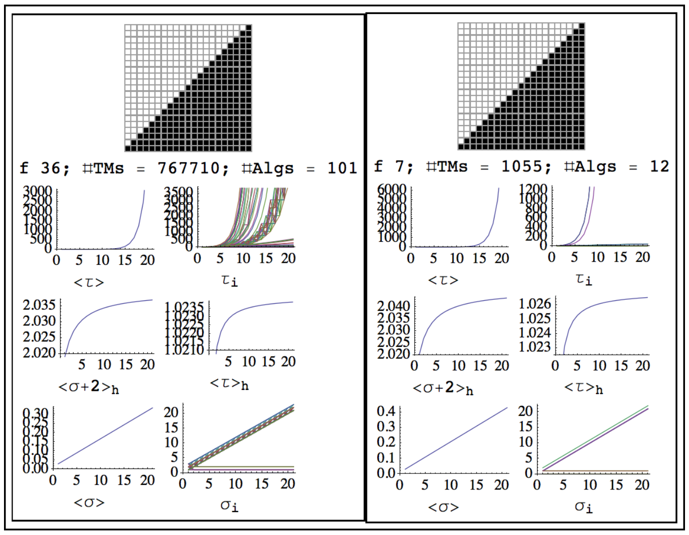


Fig. 16. Side by side comparison of an example computation of a function in (2,2) and (3,2) (the identity function).

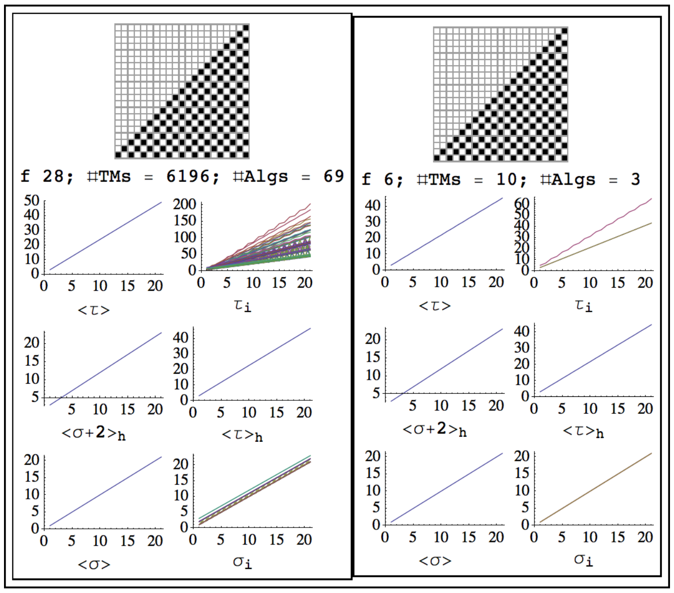


Fig. 17. Side by side comparison of the computation of a function in (2,2) and (3,2).

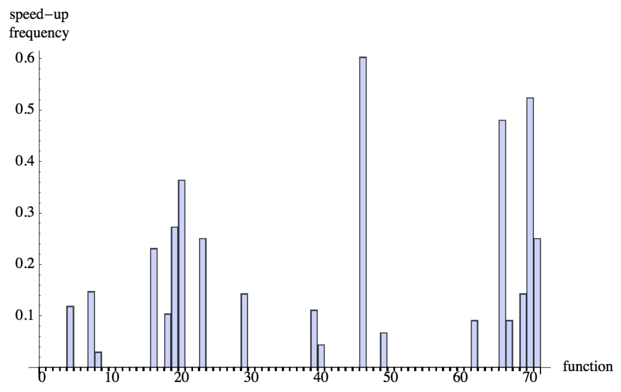


Fig. 18. Distribution of speed-up probabilities per function. Interpreted as the probability of picking a an algorithm in (3,2) computing faster an function in (2,2).

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 18 shows the scarceness of the speed-up and the magnitudes of such probabilities. Figures 19 quantify the linear factors of speed-up showing the average and maximum. The typical average speed-up was 1.23 times faster for an algorithm found when there was a faster algorithm in (3,2) computing a 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 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).

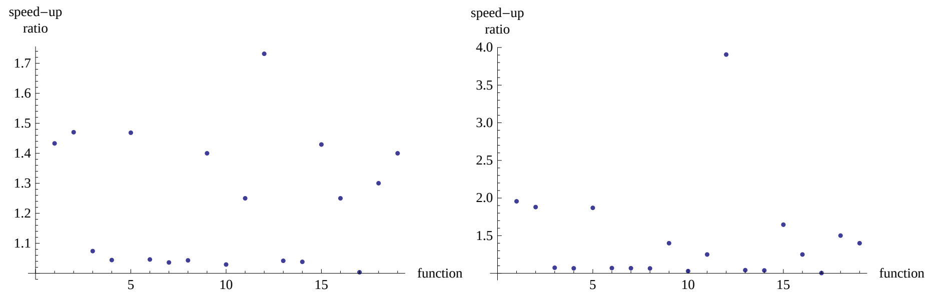


Fig. 19. Speed up significance: on the left average and on the right maximum speed-ups.

5 Concluding

We have undertaken a systematic and exhaustive study of small Turing machine with 2 colors and 2 and 3 states. For larger number of states, sampling was unavoidable and results are yet to be interpreted. The *Halting Problem* and other undecidable concerns for an experimental procedure such as the presented herein, including the problem of extensionality, were overcome by taking a finite and pragmatic approach (theory tells us that in various cases the corresponding error drops exponentially with the size of the approximation[6]). Analyzing the data gave us interesting functions with their geometrical patterns for which average and best case computations in terms of time steps were compared against descriptional complexity (the size of the machines in number of states).

Because picking an algorithm at random with uniform probability among the algorithms computing a function in an increasingly larger Turing machine space according to the maximum allowed number of states leads to increasingly greater chances to pick a slow algorithm compared to the number of fastest algorithms in the same space, one may say that an additional effort has to be made, or additional knowledge has to be known, in order to pick a faster algorithm without having to spend larger and larger resources in the search of an efficient algorithm itself. One can say that this is a *No free lunch*-type metaphor saying that speeding-up *is not for free.*

Exact evaluations with regard to runtimes and space-usages were provided shedding light onto the micro-cosmos of small Turing machines, providing figures of the halting times, the functions computed in (2,2) and (3,2) and the density of converging versus diverging computations. We found that increasing the descriptional complexity (viz. the number of *states*), the number of algorithms computing less *efficiently*, relative to the previous found runtimes in (2,2), computing a function grows faster than the number of machines more *efficiently* computing it. In other words, given a function, the set of average runtimes in (2,2) *slows down* in (3,2) with high probability.

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. Hector Zenil also wants to thank the CONACyT for providing financial support, as well as Wolfram Research.

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 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. pp. 151-158, 1971.
5. G.J. Chaitin. *Gödel's theorem and information,* Int. J. Theoret. Phys. 21, pp. 941-954, 1982.
6. C.S. Calude, M.A. Stay, *Most programs stop quickly or never halt,* Advances in Applied Mathematics, 40 295-308, 2005.
7. E. Fredkin. *Digital Mechanics,* Physica D: 254-70, 1990.
8. A. N. Kolmogorov. *Three approaches to the quantitative definition of information.* Problems of Information and Transmission, 1(1):1–7, 1965.
9. L. Levin. *Universal search problems.* Problems of Information Transmission 9 (3): 265-266. 1973
10. S. Lin & T. Rado. *Computer Studies of Turing Machine Problems.* J. ACM. 12, 196-212, 1965.
11. S. Lloyd, *Programming the Universe,* Random House, 2006.
12. S. Wolfram, *A New Kind of Science.*, Wolfram Media, 2002.
13. *Wolfram's 2, 3 Turing Machine Research Prize.*, http://www.wolframscience.com/prizes/tm23/ Accessed on June, 24, 2010.
14. R. Neary and D. Woods. *On the time complexity of 2-tag systems and small universal turing machines.* In FOCS, pages 439-448. IEEE Computer Society, 2006.
15. D. Woods & T. Neary. *Small semi-weakly universal Turing machines.* Fundamenta Informaticae. 91:161-177 (2009).