Abstract. The aim of the paper is to find tradeoffs between different measures of complexity, particularly between the concepts of geometrical, descriptional and 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, 3 and 4 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 increases as a function of the number of states, indicating that non-trivial machines tend to occupy all the resources at hand.

We did not only witness slowdown. Incidental cases of speed-up were witnessed. Throughout our study various interesting structures were found.

We present various figures disclosing these structures in the micro-cosmos of small Turing machines.

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

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 these various measures of complexity by means of computational experiments. Ultimately we also wish to include geometrical complexity measures applied to the computation configuration space.
1.1 Measures of complexity

The long run aim of the project focuses on the relationship between complexity measures, particularly computational, descriptional and geometrical 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.

**Computational Complexity** Computational complexity [4, 10] 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 a function in NP time there is 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 instance (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 [9, 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 $< M, i >$ 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 (descriptive) 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.

**Geometrical Complexity** Like with other notions of complexity, there are different measures and versions of geometrical complexity, each capturing some aspect of a geometrical object. Probably the more well-known of these measures is the Hausdorff dimension [8]. As this name already suggests, the geometrical complexity measures tend to be a generalization of the familiar concept of dimension. Thus, for example, the geometrical complexity of a line or a smooth curve is one and for a plane or a smooth surface two, and so on. In general, the geometrical complexity of an object reflects how much information is contained by a point of that object.

In the current paper we do not yet fix a particular geometrical complexity measure. However, we explore the geometrical objects that we encounter while having the characteristics and intuitions of the geometrical complexity in mind. In a later stadium of this project we wish to quantify relations between geometrical complexity and computational complexity. At that stage, it is good to bear some subtleties in mind. For almost all notions of geometrical complexity, we only obtain interesting values when the geometrical object is non-discrete. Thus, as our computations are always finite, we can only work with approximations. Moreover, most complexity notions are known to be non-computable and approximations computationally hard.

### 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, go 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.

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.
One can represent the evolution of a Turing machine by depicting the configuration of the tape at consecutive steps starting above and moving down in discrete time steps. For this particular machine, we immediately see a pattern arises. This leads us to believe that if we had more computational resources than just the 2 colors and 2 states as this Turing machine has, we can calculate the tape configuration at step \( n \) a lot faster than this machine 2506 does.

![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 [13].](image1)

![Fig. 2. Turing machine tape evolution throughout time as depicted in [13].](image2)

### 1.3 Relating notions of complexity

We want to investigate the possible relationship between the complexity of a pattern produced by the output of a Turing machine and the possibility of speeding the calculation up by adding more resources (color/states). A long run aim of this project is to scrutinize the possible thresholds.

We define a computational object as a graphical representation of some level of the computation involved. We have already seen the example above, figure 2, with the tape configuration throughout time depicted as suggested in [13]. But basically, any\(^4\) representation of some substantial part of the computation is of interest in this enterprise.

It is clear that if some substantial part of the computation has an overwhelmingly simple representation then the computation can be short-cut by exploiting this simplicity in the geometrical representation. Most likely, implementing the short-cut will take more resources (states) though. This observation proves that if some substantial part of the computation can be represented (graphically) in an easy way, then the computation is likely to be accelerable, possibly by allowing for more states in the definition of an alternative TM.

However, the other direction shall in general not hold. That is, there will be tremendously complex geometrical patterns representing some part of the computation.

\(^4\) Of course, the representation itself must be easily computable, if not it can hide lots of computational complexity in the shape of something very simple.
computation where the computation can be sped up when using more states. Worse still, we have seen very complex geometrical patterns that in the end just calculate the tape identity, that is, the computation does a lot but when it stops the tape configuration was exactly the same as when the computation started. These observations urge us to be very careful in the precise formulation of the relation between geometrical complexity and accelerability.

There are also physical reasons to believe in a strong relationship between these complexity notions. Various authors claim that “nature performs computation” [7, 12, 13] and if so, it has to order the matter that performs the computation in space and time giving rise to geometrical patterns similar to the ones we consider in our project.

We also relate and explore throughout the experiment the relationship between descriptive complexity and computational complexity. One way to increase the descriptive 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. Disregarding the answer to the P versus NP as a theoretical problem, efficient heuristics to search for the P 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 disregarding the solution in the theory. We think our approach provides insights in this regard.

1.4 Investigating the micro-cosmos of small Turing machines

We wish to see to which extent geometrical complexity is an indication to accelerability of some computational process. 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) [13, 3] and also (weak) universality in Turing machines with only 2-states and 3-colors [14].

For all practical purposes one is also restricted to perform experiments with small Turing machines (TMs) if one pursues 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. 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 Working hypotheses

The set-up of the research program of which the current paper is the first report, is centered around the following hypothesis to test.

Hypothesis A: Geometrical complexity of the tape evolution can be related to the time complexity of the corresponding algorithm.

Hypothesis B: The average runtime for a function almost always increases when allowing more resources (states).

Hypothesis C: The geometrical complexity of the fastest implementation of a function is an indication of accelerability.

We are also interested in the runtime distributions of TMs computing the same function as a possible prior distribution depending on complexity measures.

The current paper reports on the first steps in this project and the exploration of the space of small TMs. Thus, in this paper, only Hypothesis B is addressed and affirmed.

3 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.

3.1 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. Also we shall compare the respective runtimes, space-usage and corresponding geometrical patterns for the functions in the intersection of these spaces.

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, ..., 20. If a TM does not halt by 1000 steps we simply say that it diverges. Thus, we collect all the functions

5 We shall often refer to the collection of TMs with \( k \) colors and \( s \) states as a TM space.

6 Probably there is a relation like: “the larger the space complexity, the less strong the above relation”.

7 It is actually not hard to see that any function that is computed in (2,2) is also present in (3,2) whence the intersection is actually just an inclusion.
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.

### 3.2 Resources

There are \((2sk)^s\) s-state k-color one-sided tape 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 with 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\(^8\).

### 3.3 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 \(...00101\) would 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 tape “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 3.2 is undesirable.

\(^8\) Although in this case not exponentially so as halting states define no transitions.
On the basis of these considerations, and the fact that some work has been done before in the lines of this experiment\cite{13} that also contributed to motivate our own investigation, we decided to fix the TM formalism and choose the one-way tape model.

3.4 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 drawbacks:
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.

3.5 Binary output convention

None of the consideration 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. Thus, our outputs, if interpreted, will be interpreted as binary numbers. The only draw-back to this approach is that many functions will expose (at least) exponential growth. For example, the tape-identity, that is a TM that outputs the same tape configuration as the input tape configuration, will define the function $2^{x+1} - 1$. In particular, the TM that halts immediately by running off the tape while leaving the first cell black also computes the function $2^{x+1} - 1$. This is slightly undesirable but as we shall see, in our current set-up there will be few occasions where we actually wish to interpret the output as a number.
3.6 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 whether a function is defined by a particular TM and whether two TMs define the same function. It can be the case that for TMs of the size considered herein, universality is not yet attainable. This is the problem of extensionality (do two TMs define the same function) are actually decidable.

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 and we re-affirmed this in our experiments too as is shown in figure 3. 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.

3.7 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). 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 sum around 900 MB.

---

9 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)[13] and it may be (although unlikely) the case that a weak universal Turing machine in (3,2) exists.

10 Andalusian Centre for Scientific Computing.
4 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

\( \langle \text{input value}, \text{output value}, \text{runtime}, \text{space usage} \rangle \)

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.

4.1 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 4096 different TMs. Just as a matter of analogy we bring the \( C^\infty \) 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 3 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 3 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 11.
**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:

\{
{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 first 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 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 $\tau_i$ is a diagram that contains plots for all the runtimes of all the different algorithms computing the function in question. Likewise, $\sigma_i$ 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 $\sigma_i$.

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 $1/t_1 + \ldots + 1/t_n$. 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 1055 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 $\tau_i$, 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.

**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 but most of them and space-usage remain linear.
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.

**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 \(e\) such that \(W = W_e\) where \(W_e\) is defined as usual as

\[
W_e := \{ x | e(x) \downarrow \}.
\]

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 6.

**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.

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 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, ...}

TM 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 8 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 9, 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 9 (right) shows its tape
Fig. 7. Computational figures reflecting the number of available resources.

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

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.

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

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 10.

Halting probability Figure 11 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).
Fig. 9. Tape evolution for rules 2205 (left) and 1351 (right).

Fig. 10. Outputs required to characterize a function in (3,2).

Fig. 11. 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 12 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. 12. Sampling of 50 space (left) and runtime (right) sequences in (3,2).](image)

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
\[
\{\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}, \ldots
\]
that coexist with their complements, but another 54, as
\[
\{0, 3\}, \{1, 3\}, \{1, 4\}, \{0, 1, 4\}, \{0, 2, 3\}, \{0, 2, 4\}, \ldots
\]
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 13 shows, exponential runtime sequences may occur in a (3,2) function (left) with 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 894481409 steps found in the TMs number 599063 and 666364 (a pair of twin rules\(^\text{11}\)) 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:

\(^{11}\) 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. Clustering per function in (3,2).

\{14, 254, 16382, 8388606, 137438953470, \ldots \}

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 14 displays these logarithms, and the runtime and space sequences.

Fig. 14. Rule number 599063. Logarithm to base 2 of the outputs (left), runtime (center) and space usage (right).

Finally, Figure 15 shows the tape evolution with inputs 0 and 1. The pattern observed on the right repeats itself.
5 Comparison between (2,2) and (3,2)

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 general 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.

Something worth to mention is the finding of Turing machines computing the identity function in as much as exponential time as an example of a machine spending all resources to compute the simplest possible function.

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 significance of the speed-ups and slowdowns 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 3414 algorithms in (3,2), computing one of the functions in (2,2), only 122 were faster. Figure 18 shows the scarceness of the speed-up and the magnitudes of such probabilities. Figures 19 quantify the ratio of speed-up showing the average and maximum ratios. The typical average speed-up was 1.23 times faster for an algorithm found when there was a faster algorithm in (3,2) computing faster a function in (2,2).
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).
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 $2379.75 \times 10^6$ times slower than the slowest algorithm computing the same function in (2,2).

6 First conclusions and next steps

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
descriptive complexity (the size of the machines in number of states).

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 de-
scriptive 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 computing 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.

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-
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
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
157-179, 1919.
9. A. N. Kolmogorov. Three approaches to the quantitative definition of information.
265-266, 1973
11. Lin, S. and Rado, T. Computer Studies of Turing Machine Problems. J. ACM. 12,