Global Journal of Computer Science and Technology: E Network, Web & Security
Volume 16 Issue 1 Version 1.0 Year 2016
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Inductively Computable Hierarchies and Inductive Algorithmic Complexity

By Mark Burgin
University of California, United States

Abstract- Induction is a prevalent cognitive method in science, while inductive computations are popular in many fields of computer and network technology. The most advanced mathematical model of inductive computations and reasoning is an inductive Turing machine, which is natural extension of the most widespread model of computing devices and computations - Turing machine. In comparison with Turing machines, inductive Turing machines represent the next step in the development of computer science providing better models for contemporary computers and computer networks. In this paper (Section 3), we study relations between inductively computable sets, inductively recognizable sets, inductively decidable sets and inductively computable functions. In addition (Section 4), we apply the obtained results to algorithmic information theory demonstrating how inductive Turing machines allow obtaining more information for essentially decreasing complexity in comparison with Turing machines.

Keywords: algorithmic information theory, inductive computation, turing machine, inductive turing machine, kolmogorov complexity, inductive computability, inductive complexity, inductive decidability.

GJCST-H Classification: F.1.3 F.2.2

Strictly as per the compliance and regulations of:

Inductively Computable Hierarchies and Inductive Algorithmic Complexity

Mark Burgin

Abstract- Induction is a prevalent cognitive method in science, while inductive computations are popular in many fields of computer and network technology. The most advanced mathematical model of inductive computations and reasoning is an inductive Turing machine, which is natural extension of the most widespread model of computing devices and computations - Turing machine. In comparison with Turing machines, inductive Turing machines represent the next step in the development of computer science providing better models for contemporary computers and computer networks. In this paper (Section 3), we study relations between inductively computable sets, inductively recognizable sets, inductively decidable sets and inductively computable functions. In addition (Section 4), we apply the obtained results to algorithmic information theory demonstrating how inductive Turing machines allow obtaining more information for essentially decreasing complexity in comparison with Turing machines.

Keywords: algorithmic information theory, inductive computation, turing machine, inductive turing machine, kolmogorov complexity, inductive computability, inductive complexity, inductive decidability.

I. Introduction

For a long time, Turing machines dominated theoretical computer science as researchers assumed that they were absolute and all-encompassing models of computers and computations. Although Turing machines are functionally equivalent to many other models of computers and computations, such as partial recursive functions, cellular automata or random access machines (RAM), which are called recursive algorithms or recursive automata, computer scientists and mathematicians have been mostly using Turing machines for theoretical exploration of computational problems.

However, with the discovery of super-recursive algorithms and exploration of unconventional computations, more powerful models than Turing machines came to the forefront of computer science (Burgin, 2005; Burgin and Dodig-Crnkovic, 2012). One of the most natural extensions of conventional algorithmic models is inductive Turing machine, which is an innovative model of computations, algorithms and information processing systems more powerful than Turing machine. Inductive Turing machines can solve problems unmanageable by Turing machines providing means for decreasing complexity of computations and decision-making (Burgin, 2005). Consequently, in comparison with Turing machines and other recursive algorithms, inductive Turing machines represent the next step in the development of computer science as well as in the advancement of network and computational technology.

In additi on, inductive Turing machines supply more adequate than recursive algorithms and automata models of computations, algorithms, networks, and information processing systems. As a result, inductive Turing machines have found diverse applications in algorithmic information theory and complexity studies (Burgin, 2004; 2010), software testing (Burgin and Debnath, 2009; Burgin, Debnath and Lee, 2009), high performance computing (Burgin, 1999), machine learning (Burgin and Klinger, 2004), software engineering (Burgin and Debnath, 2004; 2005), computer networks (Burgin, 2006; Burgin and Gupta, 2012) and evolutionary computations (Burgin and Eberbach, 2009; 2009a; 2012). For instance, inductive Turing machines can perform all types of machine learning – TxtEx-learning, TxtFin-learning, TxtBC-learning, and TxtEx*-learning, (Beros, 2013). While the traditional approach to machine learning models learning processes using functions, e.g., limit partial recursive functions (Gold, 1967), inductive Turing machines are automata, which can compute values of the modeling functions and perform other useful operations while functions only describe such operations.

Inductive Turing machines also provide efficient tools for algorithmic information theory, which is one of the indispensable areas in information theory and is based on complexity of algorithms and automata (Chaitin, 1977; Burgin, 2010). There are different kinds and types of complexity with a diversity of different complexity measures. One of the most popular and important of them is Kolmogorov, also called algorithmic, complexity, which has turned into an important and popular tool in many areas such as information theory, computer science, software development, probability theory, and statistics. Algorithmic complexity has found applications in medicine, biology, neurophysiology, physics, economics, hardware and software engineering. In biology, algorithmic complexity is used for estimation of

Author: University of California, Los Angeles 405 Hilgard Ave.
e-mail: markburg@cs.ucla.edu

protein identification (Dewey, 1996; 1997). In physics, problems of quantum gravity are analyzed based on the algorithmic complexity of a given object. In particular, the algorithmic complexity of the Schwarzschild black hole is estimated (Dzhunushaliev, 1998; Dzhunushaliev and Singleton, 2001). Benci, et al (2002) apply algorithmic complexity to chaotic dynamics. Zurek elaborates a formulation of thermodynamics by inclusion of algorithmic complexity and randomness in the definition of physical entropy (Zurek, 1991). Gurzadyan (2003) uses Kolmogorov complexity as a descriptor of the Cosmic Microwave Background (CMB) radiation maps. Kreinovich, and Kunin (2004) apply Kolmogorov complexity to problems in classical mechanics, while Yurtsever (2000) employs Kolmogorov complexity in quantum mechanics. Tegmark (1996) discusses what can be the algorithmic complexity of the whole universe. The main problem with this discussion is that the author identifies physical universe with physical models of this universe. To get valid results on this issue, it is necessary to define algorithmic complexity for physical systems because conventional algorithmic complexity is defined only for such symbolic objects as words and texts (Li, and Vitanyi, 1997). Then it is necessary to show that there is a good correlation between algorithmic complexity of the universe and algorithmic complexity of its model used by Tegmark (1996).

In economics, a new approach to understanding of the complex behavior of financial markets using algorithmic complexity is developed (Mansilla, 2001). In neurophysiology, algorithmic complexity is used to measure characteristics of brain functions (Shaw, et al, 1999). Algorithmic complexity has been useful in the development of software metrics and other problems of software engineering (Burgin, and Debnath, 2003; Lewis, 2001). Crosby and Wallach (2003) use algorithmic complexity to study low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures.

Thus, we see that Kolmogorov/algorithmic complexity is a frequent word in present days' scientific literature, in various fields and with diverse meanings, appearing in some contexts as a precise concept of algorithmic complexity, while being a vague idea of complexity in general in other texts. The reason for this is that people study and create more and more complex systems.

Algorithmic complexity in its classical form gives an estimate of how many bits of information we need to build or restore a given text by algorithms from a given class. This forms the foundation for algorithmic information theory (Chaitin, 1977; Burgin, 2010). Conventional Kolmogorov, or recursive algorithmic complexity and its modifications, such as uniform complexity, prefix complexity, monotone complexity, process complexity, conditional Kolmogorov complexity, quantum Kolmogorov complexity, time-bounded Kolmogorov complexity, space-bounded Kolmogorov complexity, conditional resource-bounded Kolmogorov complexity, time-bounded prefix complexity, and resource-bounded Kolmogorov complexity, use conventional, i.e., recursive, algorithms, such as Turing machines. Inductive complexity studied in this paper is a special type of the generalized Kolmogorov complexity (Burgin, 1990), which is based on inductive Turing machines. It is possible to apply inductive algorithmic complexity in all cases where Kolmogorov complexity is used and even in such situations where Kolmogorov complexity is not defined. In particular, inductive algorithmic complexity has been used in the study of mathematical problem complexity (Calude, et al, 2012; Hertel, 2012; Burgin, et al, 2013), as well as for exploration of other problems (Burgin, 2010a).

The goal of this work is to find properties of inductively computable and inductively decidable sets and functions applying these properties to inductive algorithmic complexity. This paper has the following structure. In Section 2, we give definitions of simple inductive Turing machines, which can compute much more than Turing machines. In Section 3, we study relations between inductively computable sets, inductively recognizable sets, inductively decidable sets, and inductively computable functions. In Section 4, we use the obtained relations to advance inductive algorithmic complexity and algorithmic information theory. Utilization of inductive algorithmic complexity makes these relations more exact as for infinitely many objects, inductive algorithmic complexity is essentially smaller than Kolmogorov complexity (Burgin, 2004). Section 5 contains conclusion and directions for further research.

II. Simple Inductive Turing Machines as a Computational Model

Here we consider only simple inductive Turing machines (Burgin, 2005) and for simplicity call them inductive Turing machines although there are other kinds of inductive Turing machines. A simple inductive Turing machine *M* works with words in some alphabet and has the same structure and functioning rules as a Turing machine with three heads and three linear tapes (registers) – the input tape (register), output tape (register) and working tape (register). Any inductive Turing machine of the first order is functionally equivalent to a simple inductive Turing machine. Inductive Turing machine of higher orders are more powerful than simple inductive Turing machines allowing computation of more functions and sets.

The machine *M* works in the following fashion. At the beginning, an input word *w* is written in the input tape, which is a read-only tape. Then the machine *M*

rewrites the word *w* from the input tape to the working tape and starts working with it. From time to time in this process, the machine *M* rewrites the word from the working tape to the output tape erasing what was before written in the output tape. In particular, when the machine *M* comes to a final state, it rewrites the word from the working tape to the output tape and stops without changing the state.

The machine *M* gives the result when *M* halts in a final state, or when *M* never stops but at some step of the computation, the content of the output tape (register) stops changing. The computed result of *M* is the word that is written in the output tape of *M*. In all other cases, *M* does not give the result.

This means that a simple inductive Turing machine can do what a Turing machine can do but in some cases, it produces its results without stopping. Namely, it is possible that in the sequence of computations after some step, the word (say, *w*) on the output tape (in the output register) is not changing, while the inductive Turing machine continues working. Then this word *w* is the final result of the inductive Turing machine. Note that if an inductive Turing machine gives the final result, it is produced after a finite number of steps, that is, in finite time, even when the machine does not stop. So contrary to confusing claims of some researchers, an inductive Turing machine does not need infinite time to produce a result.

We assume that inductive Turing machines work with finite words in some alphabet Σ or with natural numbers represented by such words. Consequently, inductive Turing machines compute sets *X* of finite words in Σ, i.e., $X \subseteq \Sigma^*$ where Σ^* is the set of all finite words in the alphabet Σ, or sets of natural numbers $Z \subseteq \mathbf{N}$ represented by words. As it is possible to code any alphabet by words in the alphabet {0, 1}, we can assume (when it is necessary) that this binary alphabet is used by all considered inductive Turing machines.

If an inductive Turing machine *M* transforms words from Σ^* into words from Σ^*, then Σ^* is called the *domain* and *codomain* of *M*.

If an inductive Turing machine *M* transforms numbers from $\mathbf{N}$ into numbers from $\mathbf{N}$, then $\mathbf{N}$ is called the *domain* and *codomain* of *M*.

The set of words (numbers) for which the machine *M* is defined (gives the result) is called the *definability domain* of *M*.

The set of words (numbers) computed (generated) by the machine *M* is called the *range* of *M*.

III. Inductively Computable and Inductively Decidable Sets

Definition 3.1. A set $X \subseteq \Sigma^*$ or $X \subseteq \mathbf{N}$ is called *inductively computable* if there is an inductive Turing machine *M* with the range *X*.

Informally, an inductively computable set consists of all final results of some inductive Turing machine *M*.

Sets Σ^* and $\mathbf{N}$ are simple examples of inductively computable sets.

We remind that a *recursively computable* set, which is also called a *recursively enumerable* set, is the range of some Turing machine or of another recursive algorithm (Burgin, 2005).

Inductively computable sets are closely related to inductively computable functions, which have the form $f: \Sigma^* \to \Sigma^*$ or $g: \mathbf{N} \to \mathbf{N}$.

Definition 3.2. A functions *f* is called *inductively computable* if there is an inductive Turing machine *M* that computes *X*, i.e., given an arbitrary input *x*, the machine *M* computes the value *f*(*x*) when *f* is defined for *x* and does not give the result when *f* is undefined for *x*.

The domain, codomain, definability domain and range of an inductively computable function coincides with the domain, codomain, definability domain and range, respectively, of the inductive Turing machine that computes this function.

We remind that a *recursively computable* function, which is also called a *partial recursive* function, is a function computed by some Turing machine or of another recursive algorithm (Burgin, 2005).

As it is possible to simulate any Turing machine by an inductive Turing machine (Burgin, 2005), we have the following result.

Proposition 3.1. Any recursively computable function is inductively computable.

Definition 3.3. a) A set $X \subseteq \Sigma^*$ or $X \subseteq \mathbf{N}$ is called *inductively recognizable*, also called *inductively semidecidable*, if there is an inductive Turing machine *M* such that gives the result 1 for input *x* if and only if *x* belongs to *X*.

b) A set $X \subseteq \Sigma^*$ or $X \subseteq \mathbf{N}$ is called *inductively corecognizable* if there is an inductive Turing machine *M* such that gives the result 1 for input *x* if and only if *x* does not belong to *X*.

Definition 3.4. A set $X \subseteq \Sigma^*$ or $X \subseteq \mathbf{N}$ is called *inductively decidable* if there is an inductive Turing machine *M* such that gives the result 1 for any input *x* from *X* and gives the result 0 for any input *z* from $\Sigma^* \setminus X$ (from $\mathbf{N} \setminus X$).

Informally, a set *X* is inductively decidable if some inductive Turing machine *M* can indicate whether an arbitrary element belongs to *X* or does not belong. In other words, a set *X* is inductively decidable if its indication (characteristic) function is inductively computable.

Lemma 3.1. A set $X \subseteq \Sigma^*$ or $X \subseteq \mathbf{N}$ is inductively recognizable if and only if it is inductively computable.

Proof. *Sufficiency*. Let us consider an inductively computable set *X*. By definition, there is an inductive Turing machine M_X the range of which is equal to *X*. It is

possible to assume that the machine M_X gives the result if and only if its output stabilizes.

To show that X is inductively recognizable, we add a new component (subprogram) C to the machine M_X , building the new inductive Turing machine N_X . After each step of the machine M_X (as a subprogram of the machine N_X), the subprogram C checks if the two consecutive intermediate outputs of M_X are equal or not. When they are equal, the machine N_X gives the intermediate output 1and then the machine M_X (as a subprogram of the machine N_X) makes the next step.

When the two consecutive intermediate outputs of M_X are not equal, the machine N_X gives the intermediate output 1, followed by the intermediate output 0 and then the machine M_X (as a subprogram of the machine N_X) makes the next step.

This construction shows that the output of N_X stabilizes if and only if the output of M_X stabilizes. It means that the inductive Turing machine N_X gives the result 1 for any input x from X and does not gives the result otherwise. Thus, the set X is inductively recognizable.

Necessity. Let us consider an inductively recognizable set X. By definition, there is an inductive Turing machine K_X that gives the result 1 for any input x from X and either does not give the result otherwise or gives the result 0. It is possible to assume that the machine K_X gives the result if and only if its output stabilizes and all intermediate outputs of K_X are equal either to 1 or to 0.

To show that X is inductively computable, we transform the machine K_X , building the new inductive Turing machine N_X . At the beginning, K_X stores the input x. Then when the machine K_X gives the intermediate output 1, the machine N_X gives the intermediate output x. When the machine K_X gives the intermediate output 0 the first time, the machine N_X gives the intermediate output w, which is not equal to x. When the machine K_X gives the intermediate output 0 next time, the machine N_X gives the intermediate output x. Next time, the machine K_X gives the intermediate output 0, the machine N_X gives the intermediate output w and so on. Thus, even if the machine K_X obtains the result 0, the machine N_X does not give the result.

In such a way, the machine N_X obtains the result if and only if the machine K_X obtains the result. In addition, all results of N_X belong to the set X and only to it because K_X computes the indicating function of X. Thus, X is equal to the range of the function computed by N_X and consequently, X is inductively computable. Lemma is proved.

Lemma 3.2. The range of a total monotone inductively computable function is inductively decidable.

Proof. Let us consider a total monotone inductively computable function f with the range X. Then there is an inductive Turing machine M that computes f. We build an inductive Turing machine K that gives 1 as its result for all inputs from X and gives 0 as its result for all inputs that does not belong to X. It means that K decides the set X.

To achieve this goal, we include the machine M as a part (in the form of subroutine) of the machine K and define functioning of K in the following way. When K obtains a word x as the input x, the goal is to whether x belongs to the set X or does not belong. To do this, the machine K starts simulating the machine M for all inputs x_1 , x_2 , … , x_n that are less than x in a parallel mode. It means that each step is repeated for all inputs x_1 , x_2 , … , x_n , then the next step is also repeated for all inputs x_1 , x_2 , … , x_n , and so on. On each step, the machine K compares intermediate outputs of the machine M with the word x. When, at least, one of the intermediate outputs of the machine M for these inputs is equal to x, the machine K gives the intermediate output 1. When no intermediate outputs of the machine M for these inputs coincide with x, the machine K gives the intermediate output 0.

As the inductive Turing machine M computes a total function, all intermediate outputs start repeating at some step of the machine M computation. That is why the word x belong to X if on this step, it coincides with one of the outputs. By construction, the machine K continues to repeat the output 1 forever. If the word x does not coincide with any of the outputs of the machine M, then the word x does not belong to X because x can be the value of f only for arguments x_1 , x_2 , … , x_n , x as f is a monotone function.

In such a way, the machine K decides whether an arbitrary word belongs to X or does not belong.
Lemma is proved.

These lemmas allow us to prove existence of definite relations between inductively computable sets and inductively decidable sets.

Theorem 3.1. Any infinite inductively computable set contains an infinite inductively decidable subset.

Proof. Let us consider an inductively computable set X. By Lemma 3.1, the set X is inductively recognizable. It means that there is an inductive Turing machine K_X that gives the result 1 for any input x from X and either does not give the result otherwise or gives the result 0. It is possible to assume that the machine K_X gives the result if and only if its output stabilizes and all intermediate outputs of K_X are equal either to 1 or to 0 (Burgin, 2005).

As we know, there is the natural order in the set $\mathbf{N}$ and the lexicographical order in the set Σ^* (cf., for example, (Burgin, 2005)). It means that the domain of any inductive Turing machine is the ordered sequence $\{x_1 , x_2 , x_3 , \dots , x_n , \dots\}$ where $x_k < x_{k+1}$ for all $= 1, 2, 3, \dots$.

To find an inductively decidable subset in the set X, we extend the alphabet Σ by adding a new symbol

and build a new inductive Turing machine M, output of which can include this new symbol #. The machine M contains the machine K_X as a component (subroutine), a component (subroutine) G, which generates words x_1 , x_2 , x_3 , … , x_n , … one after another, and a counter C as another component (subroutine) C.

The machine M processes information in cycles organized in the following way.

Cycle 1*1: When the machine M gets the word x_1 as its input, it gives x_1 to the machine K_X , which starts processing it. At the same time, the counter C counts the number of steps made by K_X . When the machine K_X gives the intermediate output 1, the machine M gives the intermediate output x_1 , which is stored in the memory of M. When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing x_1 , the number n_1 of steps made by K_X is stored in the memory of M and the generator G generates the word x_2 .

Cycle 1*2: Then the machine M gives x_2 to the machine K_X , which starts processing it. At the same time, the counter C counts the number of steps made by K_X . When the machine K_X makes less than n_1 steps, the machine M always gives the intermediate output x_2 . After n_1 steps, when the machine K_X gives the intermediate output 1, the machine M gives the intermediate output x_2 . When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing x_2 , the number n_2 of steps made by K_X is stored in the memory of M and the machine K_X starts once more processing the word x_1 . At the same time, the counter C counts the number of steps made by K_X .

Cycle 1*3: When the machine K_X makes less than n_2 steps, the machine M always gives the intermediate output x_1 . After n_2 steps, when the machine K_X gives the intermediate output 1, the machine M gives the intermediate output x_1 . When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing x_1 , the number n_3 of steps made by K_X is stored in the memory of M and the machine K_X starts once more processing the word x_2 . At the same time, the counter C counts the number of steps made by K_X .

Cycle 1*4: When the machine K_X makes less than n_3 steps, the machine M always gives the intermediate output x_2 . After n_2 steps, when the machine K_X gives the intermediate output 1, the machine M gives the intermediate output x_2 . When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing x_2 , the number n_4 of steps made by K_X is stored in the memory of M and the generator G generates the word x_3.

Cycle 1*5: Then the machine M gives x_3 to the machine K_X , which starts processing it. At the same time, the counter C counts the number of steps made by K_X . When the machine K_X makes less than n_4 steps, the machine M always gives the intermediate output x_3 . After n_4 steps, when the machine K_X gives the intermediate output 1, the machine M gives the intermediate output x_3 . When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing x_3 , the number n_5 of steps made by K_X is stored in the memory of M and the machine K_X starts once more processing the word x_1 . At the same time, the counter C counts the number of steps made by K_X .

This process continues until it stabilizes, which happens because the definability domain of the machine K_X is not empty.

In such a way, the machine M makes the machine K_X to process more and more elements x_n , making more and more steps with each of them as its input. As the definability domain of the machine K_X is not empty, at some step m_1, the machine K_X continues forever repeating 1 as its output for an input x_k . By construction, the machine M continues forever repeating x_k as its output for an input x_1 . It means $M(x_1) = x_k$. Note that $x_1 \leq x_k$ and x_k may be not the least element in the definability domain X of the machine K_X .

Given the word x_2 as its input, the machine M performs similar cycles as before but with pairs of words (x_i , x_j).

Cycle 2*1: Thus, at the beginning when the machine M gets the word x_2 as its input, it gives x_2 and the word x_3 generated by G to the machine K_X , which starts processing both words in a parallel mode. At the same time, the counter C counts the number of steps made by K_X. When the machine K_X gives the intermediate output 1 for both inputs, the machine M gives the intermediate output x_3 , which is stored in the memory of M. When the machine K_X gives the intermediate output 0 for the input x_2 before it gives the intermediate output 0 for the input x_3 , the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_2 , x_3), the number n_1 of steps made by K_X is stored in the memory of M, the generator G generates the word x_4 and the machine M goes to the cycle 2*2.

When the machine K_X gives the intermediate output 0 for the input x_3 at the same time or before it gives the intermediate output 0 for the input x_2 , the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_2 , x_3), the number n_2 of steps made by K_X is stored in the memory of M, the generator G generates the word x_4 and the machine M goes to the cycle 2*3.

Cycle 2*2: Then the machine M gives the pair (x_3 , x_4) to the machine K_X , which starts processing it in a parallel mode. At the same time, the counter C counts the number of steps made by K_X . When the machine K_X makes less than n_1 steps, the machine M always gives the intermediate output x_4. After n_1 steps, when the

machine K_X gives the intermediate output 1 for both inputs, the machine M gives the intermediate output x_4. When the machine K_X gives the intermediate output 0 for the input x_3 before it gives the intermediate output 0 for the input x_4, the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_3, x_4), the number n_3 of steps made by K_X is stored in the memory of M and the machine M goes to the cycle 2*4. When the machine K_X gives the intermediate output 0 for the input x_4 at the same time or before it gives the intermediate output 0 for the input x_3, the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_2, x_3), the number n_2 of steps made by K_X is stored in the memory of M, the generator G generates the word x_4 and the machine M goes to the cycle 2*5.

Cycle 2*3: Then the machine M gives the pair (x_2, x_4) to the machine K_X, which starts processing it in a parallel mode. At the same time, the counter C counts the number of steps made by K_X. When the machine K_X makes less than n_2 steps, the machine M always gives the intermediate output x_4. After n_1 steps, when the machine K_X gives the intermediate output 1 for both inputs, the machine M gives the intermediate output x_4. When the machine K_X gives the intermediate output 0 for the input x_2 before it gives the intermediate output 0 for the input x_4, the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_2, x_4), the number n_2 of steps made by K_X is stored in the memory of M and the machine M goes to the cycle 2*6. When the machine K_X gives the intermediate output 0 for the input x_4 at the same time or before it gives the intermediate output 0 for the input x_3, the machine M gives the intermediate output #, the machine K_X stops processing the pair (x_2, x_3), the number n_2 of steps made by K_X is stored in the memory of M, the generator G generates the word x_4 and the machine M goes to the cycle 2*7 and so on.

This process continues until it stabilizes, which happens because the definability domain of the machine K_X is infinite.

In such a way, the machine M makes the machine K_X to process more and more pairs (x_i, x_j) functioning in a parallel mode and making more and more steps with each pair as its inputs. As in the case of the input x_1, the machine M, at first, finds the word x_k for which the machine K_X continues forever repeating 1 as its output and then locates a word $x_n > x_k$ for which the machine K_X also continues forever repeating 1 as its output. The machine M can do this because the definability domain of the machine K_X is infinite. When the machine M finds this word x_n, it continues forever repeating x_n as its output for an input x_2. It means $M(x_2) = x_n$ and $x_n > x_k$. Note that $x_2 \leq x_n$ and x_n may be not the least element in the definability domain X of the machine K_X that is larger than x_k.

Given the word x_2 as its input, the machine M performs similar cycles as before but with triples of words (x_i, x_t, x_j) as inputs to the machine K_X, which processes them in a parallel mode. In this case, the machine M, at first, finds the word x_k for which the machine K_X continues forever repeating 1 as its output and then locates a word $x_n > x_k$ for which the machine K_X also continues forever repeating 1 as its output. After this, the machine M finds the word $x_p > x_n$ for which the machine K_X continues forever repeating 1 as its output. The machine M can do this because the definability domain of the machine K_X is infinite. When the machine M finds this word x_p, it continues forever repeating x_p as its output for an input x_3. It means $M(x_3) = x_p$ and $x_p > x_n > x_k$. Note that $x_3 \leq x_p$ and x_p may be not the least element in the definability domain X of the machine K_X that is larger than x_n.

In such a way, the machine M finds results for any input x_i computing a total monotone function. By Lemma 2, the range Z of this function is inductive decidable and by construction, it is infinite.
Theorem is proved.

This result allows us to find additional properties of inductive algorithmic complexity (cf. Section 4).

Let us consider the set $R_M = \{(x, t)$; given the input x, an inductive Turing machine M gives the result in not more than t steps$\}$, i.e., R_M consists of all pairs (x, t), in which x is a word from $\{0, 1\}^*$ and t is a natural number.

Lemma 3.3. The set R_M is inductively decidable.

Proof. We build an inductive Turing machine K that gives 1 as its result for all inputs from R_M and gives 0 as its result for all inputs that does not belong to R_M. It means that K decides the set R_M.

To achieve this goal, we include the machine M as a part (in the form of subroutine) of the machine K and define functioning of K in the following way. When K obtains a word (x, t) as the input x, it starts simulating the machine M for the input x. When the step number n is made the machine K gives the intermediate output 1. Then the machine K makes one more step simulating the machine M for the input x and compares the new intermediate output of the machine M with its previous result. When these outputs coincide, the machine K gives the intermediate output 1. Otherwise the machine K gives the final output 0 and stops.

After each intermediate output 1, the machine K makes one more step simulating the machine M for the input x and compares the new intermediate output of the machine M with its previous result. When these outputs coincide, the machine K gives the intermediate output 1. As the result, the inductive Turing machine K gives 1 when the outputs of M start repeating from the step t and gives 0 as its result otherwise. In such a way, the machine K decides whether an arbitrary word (x, t) belongs to R_M or does not belong.

Lemma is proved.

Now we find additional relations between inductively computable sets and inductively decidable sets.

Taking a binary relation $R \subseteq \Sigma^* \times \Sigma^*$, it is possible to consider two projections of this relation:

The left projection $\mathrm{Pr}_l R = \{ x; \exists y ((x, y) \in R)\}$

The right projection $\mathrm{Pr}_r R = \{ y; \exists x ((x, y) \in R)\}$

Theorem 3.2. A set X is inductively computable if and only if it is the left projection of an inductively decidable binary relation.

Proof. *Necessity*. Let us consider an inductively computable set X. By definition, there is an inductive Turing machine M, which computes X.

Let us consider the set $R_M = \{(x, t)$; given the input x, an inductive Turing machine M gives the result not more than in t steps$\}$, i.e., R_M consists of all pairs (x, t), in which x is a word from $\{0, 1\}^*$ and t is a natural number. By Lemma 3.3, the set R_M is inductively decidable and $\mathrm{Pr}_l \ R_M = X$ because an element x is computed by M if and only if there is a number t such that given the input x, an inductive Turing machine M gives the result not more than in t steps.

Note that $X = \mathrm{Pr}_{lr} R^o_M$ where $R^o_M = \{(t, x); \{(x, t) \in R_M \}$ and thus, R^o_M is inductively decidable

Sufficiency. Let us consider an inductively decidable binary relation $R \subseteq \Sigma^* \times \Sigma^*$ and its left projection $\mathrm{Pr}_l R = \{ x; \exists y ((x, y) \in R)\}$, which we denote by X. By definition, there is an inductive Turing machine K_R that gives the result 1 for any input (x, y) from R and gives the result 0 for any input (z, u) that does not belong to R.

To show that the set X is inductively computable, we extend the alphabet Σ by adding the new symbol # and build a new inductive Turing machine M, which computes X. The machine M contains the machine K_R as a component (subroutine), a component (subroutine) G, which generates all words x_1 , x_2 , x_3 , … , x_n , … in the alphabet Σ one after another, and a counter C as another component (subroutine) C. The machine M processes information in cycles organized in the following way.

Cycle 1: When the machine M gets a word w as its input, the generator G produces the word x_1 and the machine M gives the pair (w, x_1) to the machine K_R , which starts processing it. At the same time, the counter C counts the number of steps made by K_R . When the machine K_X gives the intermediate output 1, the machine M gives the intermediate output w, which is stored in the memory of M. When the machine K_X gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_X stops processing the pair (w, x_1), the number n_1 of steps made by K_R is stored in the memory of M and the generator G generates the word x_2 .

Cycle 2: Then the machine M gives the pair (w, x_2) to the machine K_R , which starts processing it. At the same time, the counter C counts the number of steps made by K_R . When the machine K_R makes less than n_1 steps, the machine M always gives the intermediate output w . After n_1 steps, when the machine K_R gives the intermediate output 1, the machine M gives the intermediate output w. When the machine K_R gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_R stops processing the pair (w, x_2), the number n_2 of steps made by K_R is stored in the memory of M and the machine K_R starts once more processing the pair (w, x_1). At the same time, the counter C counts the number of steps made by K_R .

Cycle 3: When the machine K_R makes less than n_2 steps, the machine M always gives the intermediate output w . After n_2 steps, when the machine K_R gives the intermediate output 1, the machine M gives the intermediate output w . When the machine K_R gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_R stops processing the pair (w, x_1), the number n_3 of steps made by K_R is stored in the memory of M and the machine K_R starts once more processing the pair (w, x_2). At the same time, the counter C counts the number of steps made by K_R .

Cycle 4: When the machine K_R makes less than n_3 steps, the machine M always gives the intermediate output w . After n_2 steps, when the machine K_R gives the intermediate output 1, the machine M gives the intermediate output w . When the machine K_R gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_R stops processing the pair (w, x_2) , the number n_4 of steps made by K_R is stored in the memory of M and the generator G generates the word x_3 .

Cycle 5: Then the machine M gives pair (w, x_3) to the machine K_R , which starts processing it. At the same time, the counter C counts the number of steps made by K_R . When the machine K_R makes less than n_4 steps, the machine M always gives the intermediate output w . After n_4 steps, when the machine K_R gives the intermediate output 1, the machine M gives the intermediate output w . When the machine K_R gives the intermediate output 0, the machine M gives the intermediate output #, the machine K_R stops processing the pair (w, x_3), the number n_5 of steps made by K_R is stored in the memory of M and the machine K_R starts once more processing the pair (w, x_1) and this process continues, while the counter C counts the number of steps made by K_R .

This process stabilizes if and only if the machine K_R stabilizes processing a pair (w, x) for some x. If it happens, the machine M computes the word w. In this case, $w \in X$. Otherwise, w does not belong to the range of M. In this case, w also does not belong to X. As w is

an arbitrary word, it means that the machine M computes the set X.
Theorem is proved.

Corollary 3.1. A set X is inductively recognizable if and only if it is the left projection of an inductively decidable binary relation.

If R is a binary relation, then $R^o = \{(y, x); (x, y) \in R\}$ is the involution of R, also called the inverse of R.
Definitions imply the following results.

Lemma 3.4. $Pr_l R = Pr_r R^o$ and $Pr_r R = Pr_l R^o$.

Lemma 3.5. A relation R is inductively computable (inductively decidable) if and only if the relation R^o is inductively computable (inductively decidable).

Theorem 3.2 and Lemma 3.5 and 3.4 give us the following results.

Corollary 3.2. A set X is inductively computable if and only if it is the right projection of an inductively decidable binary relation.

Corollary 3.3. A set X is inductively recognizable if and only if it is the right projection of an inductively decidable binary relation.

IV. Inductive Algorithmic Complexity

Here we study inductive algorithmic complexity for finite objects such as natural numbers or words in a finite alphabet. Usually, it is the binary alphabet {0, 1}.

Definition 4.1. The *algorithmic complexity* $IC_M(x)$ of an object (word) x with respect to an inductive Turing machine M is defined as

$$AC_M(x) = \begin{cases} \min \{ l(p);\ M(p) = x\} \text{ when there is } p \text{ such that } M(p) = x \\ \text{undefined when there is no } p \text{ such that } M(p) = x \end{cases}$$

Note that if M is a Turing machine, then algorithmic complexity $AC_M(x)$ with respect to M coincides with Kolmogorov complexity $C_M(x)$ with respect to M. If M is a prefix Turing machine, then the algorithmic complexity $IC_M(x)$ is the prefix Kolmogorov complexity $K_M(x)$.

However, as in the case of conventional Kolmogorov complexity, we need an invariant complexity of objects. This is achieved by using a universal simple inductive Turing machine (Burgin, 2004; 2005).

Definition 4.2. The *inductive algorithmic complexity* $IC(x)$ of an object (word) x is defined as

$$IC(x) = \begin{cases} \min \{ l(p);\ U(p) = x\} \text{ when there is } p \text{ such that } U(p) = x \\ \text{undefined when there is no } p \text{ such that } U(p) = x \end{cases}$$

where $l(p)$ is the length of the word p and U is a universal simple inductive Turing machine.

Note that inductive complexity is a special case of generalized Kolmogorov complexity (Burgin, 1990), which in turn, is a kind of axiomatic dual complexity measures (Burgin, 2005).

The prefix inductive complexity $IK(x)$ is optimal in the class of prefix inductive complexities $IK_T(x)$.

Optimality is based on the relation $\preccurlyeq$ defined for functions $f(n)$ and $g(n)$, which take values in natural numbers:

$f(n) \preccurlyeq g(n)$ if there is a real number c such that $f(n) \leq g(n) + c$ for almost all $n \in \mathbf{N}$

Let us consider a class $\mathbf{H}$ of functions that take values in natural numbers. Then a function $f(n)$ is called *optimal* for $\mathbf{H}$ if $f(n) \preccurlyeq g(n)$ for any function $g(n)$ from $\mathbf{H}$. In the context of the axiomatic theory of dual complexities, such a function $f(n)$ is called *additively optimal* for the class $\mathbf{H}$.

Results from the axiomatic theory of dual complexities (Burgin, 1990; 2010) imply the following theorem.

Theorem 4.1. The function $IC(x)$ is optimal in the class of all prefix inductive complexities $IK_T(x)$ with respect to a prefix simple inductive Turing machine T.

As there is a simple inductive Turing machine M such that $M(x) = x$ for all words x in the alphabet {1, 0}, we have the following result.

Proposition 4.1. $IC(x)$ is a total function.

Let us assume for simplicity that inductive Turing machines are working with words in some finite alphabet and that all these words are well ordered, that is, any set of words contains the least element. It is possible to find such orderings, for example, in (Li and Vitaniy, 1997).

Theorem 4.1. If h is an increasing inductively computable function that is defined in an infinite inductively computable set W and tends to infinity when $l(x) \to \infty$, then for infinitely many elements x from W, we have $h(x) > IC(x)$.

Proof. Let us consider an increasing inductively computable function f that is defined in an infinite inductively computable set W and tends to infinity when $l(x) \to \infty$. Then by Theorem X1, W contains an infinite inductively decidable subset V. Because the set V is infinite, the restriction h of the function f on the set V tends to infinity when $l(x) \to \infty$.

By Theorem 5.3.12 from (Burgin, 2005), for infinitely many elements x from V, we have $h(x) > IC(x)$. As V is a subset of W, for infinitely many elements x from W, we have $h(x) > IC(x)$.
Theorem is proved.

Since the composition of two increasing functions is an increasing function and the composition of a recursive function and an inductively computable

function is an inductively computable function, we have the following result.

Corollary 4.1. If $h(x)$ and $g(x)$ are increasing functions, $h(x)$ is inductively computable and defined in an infinite inductively computable set W, $g(x)$ is a recursive function, and they both tend to infinity when $l(x) \to \infty$, then for infinitely many elements x from W, we have $g(h(x)) > \mathrm{IC}(x)$.

Corollary 4.2. The function $\mathrm{IC}(x)$ is not inductively computable. Moreover, no inductively computable function $f(x)$ defined for an infinite inductively computable set of numbers can coincide with $\mathrm{IC}(x)$ in the whole of its domain of definition.

As Kolmogorov complexity $\mathrm{C}(x)$ is inductively computable (Burgin, 2005), Theorem X3 implies the following result.

Theorem 4.2. For any increasing recursive function $h(x)$ that tends to infinity when $l(x) \to \infty$ and any inductively computable set W, there are infinitely many elements x from W for which $h(\mathrm{C}(x)) > \mathrm{IC}(x)$.

Corollary 4.3. In any inductively computable set W, there are infinitely many elements x for which $\mathrm{C}(x) > \mathrm{IC}(x)$.

Corollary 4.4. For any natural number a and in any inductively computable set W, there are infinitely many elements x for which $\ln_a(\mathrm{C}(x)) > \mathrm{IC}(x)$.

Corollary 4.5. In any inductively computable set W, there are infinitely many elements x for which $\ln_2(\mathrm{C}(x)) > \mathrm{IC}(x)$. If $\ln_2(\mathrm{C}(x)) > \mathrm{IC}(x)$, then $\mathrm{C}(x) > 2^{\mathrm{IC}(x)}$. At the same time, for any natural number k, the inequality $2^n > k \cdot n$ is true almost everywhere. This and Corollary X7 imply the following result.

Corollary 4.6. For any natural number k and in any inductively computable set W, there are infinitely many elements x for which $\mathrm{C}(x) > k \cdot \mathrm{IC}(x)$.

Corollary 4.7. In any inductively computable set W, there are infinitely many elements x for which $\mathrm{C}(x) > 2^{\mathrm{IC}(x)}$.

Corollary 4.8. For any natural number a and in any inductively computable set W, there are infinitely many elements x for which $\mathrm{C}(x) > a^{\mathrm{IC}(x)}$.

In addition, it is possible to apply obtained results to inductive algorithmic complexity of inductively computable functions, which are infinite objects but have a finite representation when they are enumerated.

V. Conclusion

We have found some basic properties of inductively computable, recognizable and decidable sets, as well as of inductively computable functions for computations, recognition and decision are performed by simple inductive Turing machines. These results show that inductive Turing machines form a natural extension of Turing machines allowing essentially increase power computations and decision-making.

We also applied the obtained results to algorithmic information theory demonstrating how inductive Turing machines allow obtaining more information for essentially decreasing complexity in comparison with Turing machines. The results obtained in this paper extend and improve similar results from (Burgin, 2004; 2005).

At the same time, simple inductive Turing machines form only the first level of the constructive hierarchy of inductive Turing machines (Burgin, 2005). Thus, it would be interesting to study similar properties arising in the higher levels of the constructive hierarchy. Besides, it would be useful to consider these problems in the axiomatic theory of algorithms (Burgin, 2010b).

References References References

1. Benci, V., Bonanno, C., Galatolo, S., Menconi, G. and Virgilio, M. (2002) *Dynamical systems and computable information*, Preprint in Physics cond-mat/0210654, (electronic edition: http://arXiv.org)
2. Beros, A.A. Learning Theory in the Arithmetical Hierarchy, Preprint in mathematics," math.LO/1302.7069, 2013 (electronic edition: http://arXiv.org)
3. Burgin, M. S. (1990) Generalized Kolmogorov Complexity and other Dual Complexity Measures, *Cybernetics and System Analysis*, v. 26, No. 4, pp. 481-490
4. Burgin, M. Super-recursive Algorithms as a Tool for High Performance Computing, in *Proceedings of the High Performance Computing Symposium*, San Diego, 1999, pp. 224-228
5. Burgin, M. Algorithmic Complexity of Recursive and Inductive Algorithms, *Theoretical Computer Science*, 2004, 317(1/3), pp. 31-60
6. Burgin, M. *Super-recursive Algorithms*, Springer, New York/ Heidelberg/ Berlin, 2005
7. Burgin, M. Algorithmic Control in Concurrent Computations, in *Proceedings of the 2006 International Conference on Foundations of Computer Science*, CSREA Press, Las Vegas, June, 2006, pp. 17-23
8. Burgin, M. *Theory of Information*: *Fundamentality*, *Diversity and Unification*, World Scientific, New York/London/Singapore, 2010
9. Burgin, M. (2010a) Algorithmic Complexity of Computational Problems, *International Journal of Computing & Information Technology*, v. 2, No. 1, pp. 149-187
10. Burgin, M. *Measuring Power of Algorithms, Computer Programs, and Information Automata,* Nova Science Publishers, New York, 2010b
11. Burgin, M., Calude, C.S., Calude, E. (2011) Inductive Complexity Measures for Mathematical Problems, *International Journal of Foundations of Computer Science*, v. 24, No. 4, 2013, pp. 487-500

12. Burgin, M. and Debnath, N.C. (2003) Complexity of Algorithms and Software Metrics, in Proceedings of the ISCA 18th International Conference "*Computers and their Applications*", International Society for Computers and their Applications, Honolulu, Hawaii, pp. 259-262
13. Burgin, M. and N. Debnath, Measuring Software Maintenance, in Proceedings of the ISCA 19th International Conference "*Computers and their Applications*", ISCA, Seattle, Washington, 2004, 118-121p.
14. Burgin, M. and N. Debnath, Complexity Measures for Software Engineering, *Journal for Computational Methods in Science and Engineering*, 2005, v. 5, Supplement 1, 127-143p.
15. Burgin, M. and Debnath, N. Super-Recursive Algorithms in Testing Distributed Systems, in Proceedings of the ISCA 24th International Conference "*Computers and their Applications*" (CATA-2009), ISCA, New Orleans, Louisiana, USA, April, 2009, 209-214 p.
16. Burgin, M., Debnath, N. and Lee, H. K. Measuring Testing as a Distributed Component of the Software Life Cycle*, Journal for Computational Methods in Science and Engineering*, 2009, 9(1/2), Supplement 2, 211-223p.
17. Burgin, M. and Dodig-Crnkovic, G. *From the Closed Universe to an Open World*, in Proceedings of Symposium on Natural Computing/Unconventional Computing and its Philosophical Significance**,** AISB/IACAP World Congress 2012, Birmingham, UK, July 2-6, 2012, pp. 106-110
18. Burgin, M. and E. Eberbach, Universality for Turing Machines, Inductive Turing Machines and Evolutionary Algorithms, *Fundamenta Informaticae*, v. 91, No. 1, 2009, 53-77
19. Burgin, M. and E. Eberbach, On Foundations of Evolutionary Computation: An Evolutionary Automata Approach, in *Handbook of Research on Artificial Immune Systems and Natural Computing: Applying Complex Adaptive Technologies* (Hongwei Mo, Ed.), IGI Global, Hershey, Pennsylvania, 2009a, 342-360
20. Burgin, M. and Eberbach, E. Evolutionary Automata: Expressiveness and Convergence of Evolutionary Computation, *Computer Journal*, v. 55, No. 9, 2012, pp. 1023-1029
21. Burgin, M. and Gupta, B. Second-level Algorithms, Superrecursivity, and Recovery Problem in Distributed Systems, *Theory of Computing Systems*, v. 50, No. 4, pp. 694-705, 2012
22. Burgin, M. and Klinger, A. Experience, Generations, and Limits in Machine Learning, *Theoretical Computer Science*, v. 317, No. 1/3, pp. 71-91, 2004
23. Calude, C.S., Calude, E. and Queen, M.S. (2012) Inductive Complexity of P versus NP Problem, *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, v. 7445, pp. 2-9
24. Chaitin, G.J. (1977) Algorithmic information theory, *IBM Journal of Research and Development*, v.21, No. 4, pp. 350-359
25. Crosby, S. A. and Wallach, D. S. (2003) *Denial of Service via Algorithmic Complexity Attacks*, Technical Report TR-03-416, Department of Computer Science, Rice University
26. Dewey, T .G. (1996) The Algorithmic Complexity of a Protein, *Phys. Rev. E* 54, R39-R41
27. Dewey, T .G. (1997) Algorithmic Complexity and Thermodynamics of Sequence: Structure Relationships in Proteins, *Phys. Rev. E* 56, pp. 4545-4552
28. Dzhunushaliev, V. D. (1998) Kolmogorov's algorithmic complexity and its probability interpretation in quantum gravity, in *Classical and Quantum Gravity*, v. 15, pp. 603-612
29. Dzhunushaliev, V. D. and Singleton, D. (2001) *Algorithmic Complexity in Cosmology and Quantum Gravity*, Preprint in Physics gr-qc/0108038 (electronic edition: http://arXiv.org)
30. Gurzadyan, V. G. (2003) Kolmogorov Complexity, Cosmic Background Radiation and Irreversibility, in Proceedings of XXII *Solvay Conference on Physics*, World Scientific, pp.204-218
31. Gold, E.M. Language Identification in the Limit*, Information and Control*, 1967, 10, pp. 447-474
32. Hertel, J. (2012) Inductive Complexity of Goodstein's Theorem, *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, v. 7445, pp. 141-151
33. Kraft, L.G. (1949) *A device for quantizing, grouping and coding amplitude modulated pulses*, Master's thesis, Dept of Electrical Engineering, MIT, Cambridge, Mass.
34. Kreinovich, V. and Kunin, I. A.(2004) *Application of Kolmogorov Complexity to Advanced Problems in Mechanics*, University of Texas at El Paso, Computer Science Department Reports, UTEP-CS-04-14
35. Lewis, J.P. (2001) Limits to Software Estimation, *Software Engineering Notes*, v. 26, pp. 54-59
36. Li, M. and Vitanyi, P. An Introduction to Kolmogorov Complexity and its Applications, Springer-Verlag, New York, 1997
37. Mansilla, R. (2001) *Algorithmic Complexity in Real Financial Markets*, Preprint in Physics cond-mat/0104472 (electronic edition: http://arXiv.org)
38. Shaw, F.Z., Chen, R.F., Tsao, H.W. and Yen, C.T. (1999) Algorithmic complexity as an index of cortical function in awake and pentobarbital-anesthetized rats, *J. Neurosci. Methods*, 93(2) 101-110
39. Tegmark, M. (1996) Does the Universe in Fact Contain almost no Information? *Found. Phys. Lett.*, v. 9 pp. 25-42

40. Yurtsever, U. (2000) Quantum Mechanics and Algorithmic Randomness, *Complexity*, v. 6, No.1, pp.27–34
41. Zurek, W. H. (1991) Algorithmic information content, Church-Turing thesis, physical entropy, and Maxwell's demon, in *Information dynamics (Irsee, 1990),* NATO Adv. Sci. Inst. Ser. B Phys., 256, *Plenum, New York*, pp. 245-259

Global Journals Inc. (US) Guidelines Handbook 2016

www.GlobalJournals.org