Artificial Intelligence and Computer Science

Contributors

Shotaro Akaho
Wah-Sui Almberg
Jordi Atserias
Magnus Boman
Mark Burgin
Mehdi Dastani
Andrzej Karbowski
A. Kolakowska
Ingo Kreuz
Viacheslav Novikov
M. A. Novotny
Dieter Roller
Leendert van der Torre

Susan Shannon
Editor

NOVA

TP18
A791.5

ARTIFICIAL INTELLIGENCE AND COMPUTER SCIENCE

SUSAN SHANNON
EDITOR

Nova Science Publishers, Inc.
New York

LIBRARY OF CONGRESS CATALOGING-IN-PUBLICATION DATA

Artificial Intelligence and Computer Science / Susan Shannon, editor.
p. cm.
Includes index.
ISBN 1-59454-411-5 (hardcover)
1. Artificial Intelligence and Computer Science. Susan Shannon.
QC173.7.T74 2005
530.14'072--dc22 2004024758

Published by Nova Science Publishers, Inc. ✢ New York

ARTIFICIAL INTELLIGENCE AND COMPUTER SCIENCE

PREFACE

We are living in a world where complexity of systems created and studied by people grows beyond all imaginable limits. Computers, their software and their networks are among the most complicated systems of our time. Science is the only efficient tool for dealing with this overwhelming complexity. One of the methodologies developed in science is the axiomatic approach. It proved to be very powerful in mathematics. In Chapter 1, we develop further an axiomatic approach in computer science initiated by Floyd, Manna, Blum and other researchers. In the traditional constructive setting, different classes of algorithms (programs, processes or automata) are studied separately, with some indication of relations between these classes. In such a way, the constructive approach gave birth to the theory of Turing machines, theory of partial recursive functions, theory of finite automata, and other theories of constructive models of algorithms. The axiomatic context allows one to research collections of classes of algorithms, automata, and processes. These classes are united in a collection by common properties in a form of axioms. As a result, axiomatic approach goes higher in the hierarchy of computer and network models, reducing in such a way complexity of their study. The suggested axiomatic methodology is applied to evaluation of possibilities of computers and their networks. People more and more rely on computers and other information processing systems. So, it is vital to know better than we know now what computers and other information processing systems can do and what they can't do. The main emphasis in this paper is done on such properties as computability, decidability, and acceptability. These properties are important for the further development of artificial intelligence.

Chapter 2 reviews current algorithms for distributed, asynchronous control of data networks. Different problem formulations are considered: from the simplest shortestpath approach, without Quality of Service (QoS) constraints, via total flow cost minimization for given traffic quality equations, until dynamic flow control with influencing users through transmission prices. These different formulations are presented in a unified way and compared from the possible application area point of view.

In order to solve complex configuration tasks in technical domains, various knowledge based methods have been developed. However, their applicability is often unsuccessful due to their low efficiency. One of the reasons for this is that (parts of the) problems have to be solved again and again, instead of being "learnt" from preceding processes. However, learning processes bring with them the problem of conservatism, for in technical domains *innovation* is a deciding factor. On the other hand, a certain amount of conservatism is often desired since uncontrolled innovation as a rule is also detrimental.

Chapter 3 proposes the heuristic RKF (Relevant Knowledge First) for making decisions in configuration processes, based on the so-called *relevance* of objects in a knowledge base. The underlying relevance-function has two components, one based on reinforcement learning and the other based on forgetting (fading). The relevance of an object increases with its successful use and decreases with age, when it is not used. RKF has been developed to speed up the configuration process and to improve the quality of the solutions in relation to the user's rating of the solutions.

An algorithm for managing a portfolio of stocks using a trading agent is presented in Chapter 4. A simulation game inspired by history-based Parrondo games is described. A performance measure is defined, with which various strategy mixes can be judged. Even when transaction costs are taken into account, active portfolio management (as opposed to Buy and Hold) is shown to be profitable.

In Chapter 5 the authors propose a novel criterion for support vector machine (SVM) learning: maximizing the margin in the input space, not in the feature space like the original SVM. This criterion is appropriate in particular when some prior knowledge is given by a metric in the input space. We derive an algorithm that consists of two alternating steps to estimate the dual sets of parameters, where those parameters are first initialized by the original SVM. The first step is to update one set of parameters by a Newton-like procedure, and the second step is to update the dual set by solving a quadratic programming (QP) problem. The algorithm converges to a local optimum in a few steps under mild conditions, and it preserves the sparsity of support vectors. Although the complexity for calculating temporal variables becomes large, the complexity for the QP problem does not change. It is also shown that the original SVM can be seen as a special case of this framework. We further derive a simplified algorithm which enables us to use existing codes for the original SVM.

In a state-update protocol for a system of L asynchronous parallel processes that communicate only with nearest neighbors, global desynchronization in operation times can be deduced from kinetic roughening of the corresponding virtual-time horizon (VTH). The utilization of the parallel processing environment can be deduced by analyzing the microscopic structure of the VTH. In Chapter 6 we give an overview of how the methods of non-equilibrium surface growth (physics of complex systems) can be applied to uncover some properties of state update algorithms used in distributed parallel discrete-event simulations (PDES). In particular, we focus on the asynchronous conservative PDES algorithm in a ring communication topology. The time evolution of its VTH is simulated numerically as asynchronous cellular automaton whose update rule corresponds to the update rule followed by this algorithm. There are two cases of a balanced load considered: (1) the case of the minimal load per processor, which is expected to produce the lowest utilization (the so-called worst-case performance scenario); and, (2) the case of a general finite load per processor. In both cases, we give theoretical estimates of the performance as a function of L and the load per processor, i.e., approximate formulas for the utilization (thus, the mean speedup) and for the desynchronization (thus, the mean memory request per processor). It is established that the memory request per processor, required for state savings, does not grow without limit for a finite number of processors and a finite load per processor but varies as the conservative PDES evolve. For a given simulation size, there is a theoretical upper bound for the desynchronization and a theoretical non-zero lower bound for the utilization. We show that the conservative PDES are generally scalable in the ring communication topology. The new approach to performance studies, outlined in this chapter, is particularly useful in the search for the design of a new-generation of algorithms that would efficiently carry out an autonomous or tunable synchronization.

Chapter 7 explores a new robust approach for Semantic Parsing of unrestricted texts. Our approach formalizes Semantic Parsing as a Consistent Labelling Problem (CLP), allowing the integration of several knowledge types (syntactic and semantic) obtained from different sources (linguistic and statistical). The current implementation obtains 95% accuracy in model identification and 72% in case-role filling.

In Chapter 8 the description of the language model of data, offered by the author, responding the purposes of storage and retrieval of complicatedly structured data and initiation of control operations, according to recursive rules, having the same structure, as the data, is given. In the model, as it is accepted, the structural, manipulation and integrity parts are allocated, the features of implementation are considered. The possibilities of formulation and realization of the tasks of teaching and semantic search of information on natural language within the framework of the model are discussed also.

In Chapter 9 we investigate the relation between decisions, deliberation and agent types. In particular, we are interested how deliberation leads to decisions, and how agent types classify patterns of deliberation. We therefore consider Classical and Qualitative Decision Theories (CDT and QDT), the Beliefs-Desire-Intention (BDI) model, 3APL systems, and Belief-Obligation-Intention-Desire (BOID) systems. The first two are based on a decision rule which expresses a notion of rationality, whereas the latter three are based on deliberation processes and agent types.

Contents

Chapter 1	Measuring Power of Algorithms, Programs and Automata *Mark Burgin*	**1**
Chapter 2	Distributed, Asynchronous Algorithms for Data Networks Control - A State of the Art Review *Andrzej Karbowski*	**63**
Chapter 3	Reinforcement Learning and Forgetting for Knowledge Based Configuration *Ingo Kreuz and Dieter Roller*	**87**
Chapter 4	An Active Agent Portfolio Management Algorithm *Wah-Sui Almberg and Magnus Boman*	**127**
Chapter 5	SVM That Maximizes the Margin in the Input Space *Shotaro Akaho*	**139**
Chapter 6	Desynchronization and Speedup in an Asynchronous Conservative Parallel Update Protoco *A. Kolakowska and M. A. Novotny*	**155**
Chapter 7	A Robust Semantic Parsing Approach *Jordi Atserias*	**181**
Chapter 8	Language Model of Data and Some Features of Its Implementation *Viacheslav Novikov*	**201**
Chapter 9	Decisions, Deliberation, and Agent Types CDT - QDT - BDI - 3APL - BOID *Mehdi Dastani and Leendert van der Torre*	**217**
Index		**235**

In: Artificial Intelligence and Computer Science
Editor: Susan Shannon, pp. 1-61

ISBN: 1-59454-411-5

Chapter 1

MEASURING POWER OF ALGORITHMS, PROGRAMS AND AUTOMATA

Mark Burgin
Department of Mathematics, University of California, Los Angeles
405 Hilgard Ave., Los Angeles, CA 90095

Abstract

We are living in a world where complexity of systems created and studied by people grows beyond all imaginable limits. Computers, their software and their networks are among the most complicated systems of our time. Science is the only efficient tool for dealing with this overwhelming complexity. One of the methodologies developed in science is the axiomatic approach. It proved to be very powerful in mathematics. In this paper, we develop further an axiomatic approach in computer science initiated by Floyd, Manna, Blum and other researchers. In the traditional constructive setting, different classes of algorithms (programs, processes or automata) are studied separately, with some indication of relations between these classes. In such a way, the constructive approach gave birth to the theory of Turing machines, theory of partial recursive functions, theory of finite automata, and other theories of constructive models of algorithms. The axiomatic context allows one to research collections of classes of algorithms, automata, and processes. These classes are united in a collection by common properties in a form of axioms. As a result, axiomatic approach goes higher in the hierarchy of computer and network models, reducing in such a way complexity of their study. The suggested axiomatic methodology is applied to evaluation of possibilities of computers and their networks. People more and more rely on computers and other information processing systems. So, it is vital to know better than we know now what computers and other information processing systems can do and what they can't do. The main emphasis in this paper is done on such properties as computability, decidability, and acceptability. These properties are important for the further development of artificial intelligence.

Key words: computation, computing power, acceppting power, axiom, solvability, computability, decidability, acceptability

1 Introduction

Mathematical methods play more and more important role in society. Mathematics is applied to a diversity of other fields. Mathematics provides a variety of methods for description, modeling, computation, reasoning, constructing, etc. This extensive variety of methods is traditionally divided into two directions: constructive and axiomatic. Beginning from Euclid's "*Elements*," which present geometry as an axiomatic discipline, axiomatic methods have demonstrated their power in mathematics. As Burton (1997) writes, generation after generation regarded the "*Elements*" as the summit and crown of logic and mathematics, and its study as the best way of developing facility of exact reasoning. Abraham Lincoln at the age of forty, while still a struggling lawyer, mastered the first six books of Euclid, solely as training for his mind. Even now, in spite of the discovery of non-Euclidean geometries and improvements of the system of Euclid, "Elements" largely remains the supreme model of a book in mathematics, demonstrating the power of the axiomatic approach.

The main goal of this paper is to show that axiomatic methods are also very efficient for computer science. When computers were created and utilized, the theory of algorithms formed a core of computer science and till now this is the most developed mathematical discipline of computer science. Methods and technique developed in this paper for the theory of algorithms and computation are oriented at various applications in computer science and technology. Examples of practical problems that benefit from this approach are debugging and testing computer software, design of software metrics, and comparison of computational power for different systems.

It is possible to consider three levels of axiomatization: local, global, and multiglobal.

Local or *object oriented axiomatization* gives axioms for a description/determination of a single object, e.g., a separate program or algorithm. As an example, we can take the axiomatic theory of programs originated in (Naur, 1966; Floyd, 1967) and developed further in (Hoare, 1969; Milner, 1978) and many other books and papers.

Global or *class oriented axiomatization* gives axioms for a description/determination of a definite class of objects, e.g., Euclidean geometry, the class of all vector spaces, and the class of all groups. Axiomatic definitions of programming languages give other examples of global or class oriented axiomatization (Hoare and Wirth, 1973; Schwartz, 1978; Meyer and Halpern, 1980).

Multiglobal or *feature oriented axiomatization* gives axioms for a description/determination of a system of definite classes that have some specific features in common, e.g., all set theories in which the Axiom of Choice is valid or all classes of finite groups.

In the global approach, a system of axioms is taken (built or selected) for a description of some system. Then these axioms are used for deduction of properties of this system. This is the standard mathematical way of axiomatic studies. It is embodied in such classical works as Euclid's *Elements*, Hilbert's axiomatization of the Euclidean geometry, and axiomatic set theories (cf. Fraenkel and Bar-Hillel, 1958): **ZF** of Zermelo-Fraenkel, **VN** of von Neumann, **BG** of Bernays-Godel, and two theories of Quine – **NF** and **ML**. The problem that is solved by a global axiomatization is what are basic properties of a given system that allow one to deduce all other properties.

In contrast to this, the multiglobal approach is oriented not at a system of objects but at definite properties of such systems. The aim of multiglobal axiomatization is to characterize by simple properties P_1 , … , P_m such classes of systems in which some important results R_1 , … , R_n are valid. Properties P_1 , … , P_m are formulated as axioms and conditions A_1 , … , A_m and the necessary results R_1 , … , R_n are deduced from these axioms and conditions as theorems T_1 , … , T_n . Usually, the initial properties P_1 , … , P_m in the form of axioms and conditions A_1 , … , A_m are transparent and easy to test, while the deduced properties R_1 , … , R_n are complicated and hidden from direct comprehension.

This allows one not to prove the same result, for example, R_i , for each class from a diverse variety of classes separately, but to check for each of those classes only initial axioms, which are much simpler, and then to deduce this result R_i from a general theorem T_i proved in the axiomatic setting. Axiomatic form of concept introduction in modern mathematics, such as rings, fields, Hilbert spaces, Banach spaces etc., gives classical examples of such a technique. Another example of this approach is Blum's (1967) concept of the size of a machine, which synthesized many measures of algorithms and software metrics. Another his concept is computational complexity, which in the axiomatic form synthesized such concepts as time complexity and space complexity of computation (Blum, 1967a). The further development of this approach resulted in axiomatic development of the theory of complexity of algorithms and computation (Burgin, 1982; 1992) and its application to software measures (Burgin and Debnath, 2003).

The first Godel's incompleteness theorem (1931) shows that the problem of global axiomatization cannot be completely solved for sufficiently rich mathematical systems. Axioms cannot present all properties of such systems, given traditional means of inference. Thus, it becomes more efficient to consider partial systems of axioms that define systems of classes in the context of multiglobal axiomatization. According to the modern methodology of science, such systems of axioms become laws of the second order (Burgin and Kuznetsov, 1994).

The reason why we need multiglobal axiomatization in computer science is existence of a huge diversity of computer and network systems, programs and program systems, as well as their theoretical models. Axiomatics allows one to compress information about all these devices and systems, providing efficient means for their study and development.

An extreme case of the multiglobal axiomatization is *reverse mathematics* aimed at finding minimal conditions for a possibility to prove some result or a system of results. Multiglobal axiomatization is also closely related to reverse mathematics, which strives to obtain the simplest conditions under which a given result (mathematical theorem) is valid. Namely, reverse mathematics is the branch of mathematics concerned with what are the minimal axioms needed to prove the particular theorem (Friedman and Hirst, 1990; Giusto and Simpson, 2000). It turns out that over a weak base theory, many mathematical statements are equivalent to the particular additional axiom needed to prove them.

Axiomatic approach brings the theory of algorithms to a new level. Historically, this theory appeared when models of algorithms were constructed to prove absolute undecidability of some mathematical problems. Such ultimate undecidability demanded to show that there is no algorithm for problem solution. In other words, it was necessary to give an exact description of all possible algorithms. The goal was achieved by constructing such ultimate classes of recursive algorithms as Turing machines, partial recursive functions, and many other models of algorithms. However, recent development of computer science demonstrated that these classes are not absolute and there are more powerful algorithms (cf., for example, Burgin, 2001). Thus, we have come to the situation when it is impossible to reduce decidability problems to one universal class of algorithms. Axiomatic setting allows one to eliminate this obstacle and to prove undecidability without constructive models of algorithms. Many other properties are also obtained from axioms and simple conditions.

In this paper, we consider the most conventional class of reactive algorithms and abstract automata. These algorithms and automata react to a given input by producing some output or coming to some state. Active, interactive, and proactive algorithms and automata are considered elsewhere.

Our advent in a new realm of multiglobal axiomatics of algorithms, automata, and programs, we begin (Section 2) with an analysis of such basic for computer science concepts as algorithms, programs, and abstract automata. The suggested approach allows us to eliminate some contradictions and inconsistencies in the conceptual system of computer science.

In Section 3, types of algorithms and functioning of automata and algorithms are analyzed and formalized. A multifaceted typology is developed for algorithms aimed at axiomatizing computer science.

In Section 4, basic axioms for algorithms are considered in three forms: *postulates* as the most basic assumptions, *axioms* as global assumptions that represent important properties, and *conditions* as local assumptions that represent specific properties.

An example of computer science postulates is the Deterministic Computation Postulate **PDC**, which states that any algorithm A that takes inputs from a set X and gives outputs that belong to a set Y determines a function f_A from X into Y.

An example of computer science axioms is the Universality Axiom **AU**, which states that for a class **K** of automata/algorithms and some coding $\mathbf{c} : \mathbf{K} \rightarrow V^+$, there is a universal algorithm/automaton in **K**.

An example of computer science conditions is the Switching Condition **SW**, which states that for any x and y from X, there is a switching for x and y algorithm/automaton in a class **K**.

In Section 5, power of algorithms and automata is investigated. Exact mathematical methods are developed for comparison and evaluation of algorithms and automata.

In Section 6, properties and related problems of algorithms and automata are classified and studied. All properties and problems are separated into several classes: linguistic, functional, description, operational, etc. Some of these classes have been intensively studied for different models of algorithms, such as finite automata, Turing machines, and some others. Here these properties and problems are considered in axiomatic setting, allowing one to essentially expand the scope of applications. Other studied here properties and problems have been beyond the scope of conventional computer science although they are important for practice.

In Section 7, boundaries for algorithms and computation are found. In particular, it is proved that all non-trivial linguistic (Theorem 7.5) and functional (Theorem 7.6) properties are undecidable for a lot of classes of algorithms. This implies classical results of Rice (1951) for Turing machines and some results from (Adleman and Blum, 1991) for inductive inference. At the same time, for other kinds of properties (e.g., operational properties), there are both decidable and undecidable non-trivial properties.

Applications of the theoretical results to software and hardware verification and testing are considered in Section 8.

Denotations and Basic Definitions:

$\boldsymbol{N}$ is the set of all natural numbers;

ω is the sequence of all natural numbers;

$\varnothing$ is the *empty set*;

The logical symbol $\forall$ means "for any";

The logical symbol $\exists$ means "there exists";

If **P** is a property, then $\neg$**P** is the *complementary property*, i.e., an object x has the property **P** if and only if it does not have the property $\neg$**P**.

$\boldsymbol{R}$ is the set of all real numbers;

A *binary relation* T between sets X and Y is a subset of the direct product $X{\times}Y$. The set X is called the *domain* of T ($X = \mathrm{D}(T)$) and Y is called the *codomain* of T ($Y = \mathrm{CD}(T)$). The *range* of the relation T is $\mathrm{R}(T) = \{ y ; \exists\, x \in X\ ((x, y) \in T) \}$. The *definability domain* of the relation T is $\mathrm{DD}(T) = \{ x ; \exists\, y \in Y\ ((x, y) \in T) \}$.

A *function* or *total function* from X to Y is a binary relation between sets X and Y in which there are no elements from X which are corresponded to more than one element from Y and to any element from X is corresponded some element from Y. Often total functions are also called everywhere defined functions.

A *partial function* f from X to Y is a binary relation in which there are no elements from X which are corresponded to more than one element from Y.

For a partial function f, its *definability domain* $\mathrm{DD}(f)$ is the set of all elements for which f is defined.

A function $f_{\varnothing}$ is called *empty* or *nowhere defined* if $\mathrm{DD}(f) = \varnothing$.

For any set, S, $\chi_S(x)$ is its *characteristic function*, that is, $\chi_S(x)$ is equal to 1 when $x \in S$ and is equal to 0 when $x \notin S$, and $C_S(x)$ is its partial characteristic function, that is, $C_S(x)$ is equal to 1 when $x \in S$ and is undefined when $x \notin S$.

An *alphabet* or vocabulary A of a *formal language* is a set consisting of some symbols or letters. A vocabulary is an alphabet on a higher level of hierarchy because words of a vocabulary play the same role for building sentences as symbols in an alphabet for building words. Traditionally any alphabet is a set. However, a more consistent point of view is that an alphabet is a multiset (Knuth, 1981), containing an unbounded number of identical copies of each symbol.

A *string* or *word* is a sequence of elements from the alphabet. A^* denotes the set of all finite words in the alphabet A. Usually there is no difference between strings and words. However, having a language, we speak about words of this language and not about its strings. A^{**} denotes the set of all (finite and infinite) words in the alphabet A. A^+ denotes the set of all non-empty finite words in the alphabet A. A^{++} denotes the set of all non-empty (finite and infinite) words in the alphabet A.

A *formal language* L is any subset of A^*.

The *length* $\mathrm{l}(w)$ of a word w is the number of letters in the word w.

ε is the *empty word*.

Λ is the *empty symbol*.

When an algorithm/automaton A gives y as the result being applied to x, it is denoted by $A(x) = y$. When an algorithm/automaton A gives no result being applied to x, it is denoted by $A(x) = *$.

D(A) is the *domain* of an algorithm A, i.e., the set X of such elements that are processed by A.

DD(A) is the *definability domain* of an algorithm A, i.e., is the set X of such elements that if any of them is given as input to A, then A gives a result/output.

C(A) is the *codomain* of an algorithm A, i.e., the set Y of such elements that are tentative outputs of A.

2 Algorithms, Programs, Procedures, and Abstract Automata

We begin with clarification of relations between such basic concepts of computer science as algorithms, programs, and automata.

In many cases, these terms are used interchangeably. However, computing practice and research experience show that these concepts are different. For instance, programmers and computer scientists are aware that the same algorithm can be expressed in a variety of forms. Texts are conventional forms of algorithm representation. For many algorithms, a representing text can be practically in any language, from natural languages like English or Spanish to programming languages like C^{++} or Java. That is why, as it is stressed by Shore in (Buss *et al*, 2001), it is important to know that algorithm is different from its representation and to make a distinction between algorithms and their representations. In a similar way, Cleland (2001) emphasizes that "it is important to distinguish instruction-expressions from instructions." Instructions are the simplest algorithms. The same instruction may be expressed in many different ways, including in different languages and in different terminology in the same language. Also, some instruction may be communicated non-verbally, e.g., when one computer sends a program to another computer in a form of electrical signals.

However, advanced languages provide more means for algorithm representation. As a result, not all algorithms that can be described in English have a representation Zulu. It would be an interesting problem to compare natural languages by those means that they provide for algorithm representation. The same is true for mathematical models of algorithms. As we know Turing machines allow one to represent much more algorithms than finite automata. For instance, addition of arbitrary binary numbers can be represented in many models: by a Turing machine, by a formal grammar, by a neural network, or by a finite automaton. At the same time, it is possible to realize multiplication of arbitrary binary numbers by a Turing machine, neural network or formal grammar. No finite automaton can multiply arbitrary binary numbers.

In addition, an algorithm can be represented not only in a symbolic form but also by hardware. For instance, in computer memory, programs and consequently, algorithms, are represented by states of memory cells. Another type of algorithm representation by hardware gives digital arithmetic. Digital arithmetic encompasses study of symbolic number representations, algorithms for operations on numbers, implementations of arithmetic units in hardware, and their use in general-purpose and application-specific systems (Ercegovac, M.D. and Lang, 2002; Wakerly, 2001). Numerical algorithms are represented by (embodied in) integrated circuits of arithmetical units (processors) of computers.

Moreover, neural networks realize dynamic storage of information, utilizing a part of the network. To preserve its state, a neuron can be initiated to go into a self-sustaining loop that keeps the neuron firing even when the top input is no longer active. The stored binary bit is continuously accessible by looking at the output. This configuration is called a latch. The part of a network that used only for dynamical storage plays the role of the short-term memory of people or primary memory of computers: random access memory (RAM) and read only memory (ROM). Experimental evidence shows existence of such a dynamic memory in the brain (Suppes and Han, 2000). When we store a program in this dynamic memory, we have a representation of an algorithm by a process.

Consequently, we come to two forms of algorithm representation: *iconic*, *symbolic*, and *physical*.

Definition 2.1. *An algorithm representation by means of structures is called iconic.*

A neural network or a cellular automaton in graphical form is an example of an iconic algorithm representation.

Definition 2.2. *An algorithm representation by means of symbols is called symbolic or an algorithm description.*

A program printed on paper is an example of a symbolic algorithm representation.

Definition 2.3. *An algorithm representation by means of a physical entity (hardware) is called physical.*

A program stored in computer memory is an example of a system algorithm representation.

In turn, physical algorithm representation can be subdivided into three classes: *system*, *state*, and *process* physical algorithm representation.

Chips of arithmetic units form system algorithm representation. A program written in a CD is a state algorithm representation. Specific processes in the brain form process algorithm representation.

It is also possible to divide physical representations by types of systems and processes. It gives us mechanical, electronic, and quantum representations of algorithms. Here it is necessary to remark that in the case of information, it is more correct to write about quantum representation of information and not about quantum information.

There are three main classes of symbolic representations for algorithms and procedures:

Automaton representations. Turing machines and finite automata give the most known examples of such representations.

Instruction representations. Formal grammars, rules for inference, and Post productions give the most known examples of such representations.

Equation representations. Here an example of such recursive equation is given.

$$\text{Fact}\,(n) = \begin{cases} 1 & \text{when} \quad n = 1, \\ n \cdot \text{Fact}\,(n - 1) & \text{when} \quad n > 1 \end{cases}$$

The fixed point of this recursive equation defines a program for computation of the factorial $n!$

We frequently encounter similar situations of multiple representations in nature and society. For instance, it emerges in the case of numbers. For instance, the same rational number may be represented by the following fractions 1/5 , 2/10 or 3/15, as well as by decimals 0.2, 0.20 or 0.19999... Number seven is represented by the Arab (or more exactly, Hindu) numeral 7 in the decimal system, the sequence 111 in the binary number system, and by three symbols VII in the Roman number system.

There are, at least, three natural ways for separation of algorithms from their descriptions such as programs or systems of instructions.

The model approach:

Some type **D** of algorithm representations (for example, Turing machines) is taken as a model description, in which there is a one-to-one correspondence between algorithms and their representations. Then we introduce an equivalence relation R between different representations of algorithms. This relation has to satisfy two axioms:

(DA1) Any representation of an algorithm is equivalent to some element from the model class **D**.

(DA2) Any two elements from the model class **D** belong to different equivalence classes.

Such an approach is applied by Moschovakis, who considers the problem of unique representation for algorithms in his paper "What is an Algorithm?' (2001). He demonstrates that machine models of algorithms, such as Turing machines, are only models but not algorithms themselves. The main argument is that there are many models for one and the same algorithm. To remedy this, he defines algorithms as systems of mappings, building insuch a way a new model for algorithm, which is defined by recursive equations and called *recursor*. This is a progress in understanding and mathematical modeling algorithms. However, this does not solve the problem of separating algorithms as something invariant from their representations. This type of representation is on higher level of abstraction than traditional representations, such as Turing machines or partial recursive functions. Nevertheless, a *recursor* (in the sense of Moschovakis) is only a new model for algorithm but not algorithm itself.

The relational approach:

An equivalence relation R between different descriptions of algorithms is chosen. With this relation, we define algorithm as a class of equivalent descriptions. Equivalence of descriptions can be determined by some natural axioms, describing, for example, properties of operations:

Composition Axiom. Composition (sequential, parallel, etc.) of descriptions represents the corresponding composition of algorithms.

Decomposition Axiom. If a description H represents a sequential composition of algorithms A and B, a description K represents a sequential composition of algorithms C and B, and $A = C$, then H is equivalent to K.

At the same time, the equivalence relation R between descriptions can be formed on the base of computational processes. Namely, two descriptions define the same algorithm if these descriptions generate the same sets of computational processes. This definition depends on our understanding of equal processes. For example, in some cases it is natural to consider processes on different computing devices as different, while in other cases it might be better to treat some processes on different computing devices as equal.

An example of such a relation R is obtained by modification of the rule suggested by Cleland (2001) for instructions:

Different algorithm descriptions, i.e., representations of algorithms, express the same algorithm only if they prescribe the same type of functioning.

Such a definition of algorithm is not unique and depends on organization of computational processes. For example, let us consider some Turing machine **T** and another Turing machine **Q**. The only difference between **T** and **Q** is that **Q** contains all instruction of **T** and one more instruction that is never used in computations of the machine **Q**. Then, on one hand, it is possible to assume that this additional instruction has no influence on computational processes and thus, **T** and **Q** *algorithm* one and the same algorithm. On the other hand, if a Turing machine in a course of computation always go through all instructions to choose the one to be performed, then the processes are different and consequently, **T** and **Q** represent different algorithms.

The structural approach:

In it, a specific invariant (structure) is extracted from different representations. This structure is called an algorithm. Here we understand structures in the sense of (Burgin, 1997). It results in the following understanding, which separates algorithm from its descriptions.

Definition 2.4. *An algorithm is a (finite) structure with exact effective information (instructions) for some performer (class of performers) that allows this performer(s) to perform operations (actions) in order to achieve a definite goal.*

This shows that algorithms are compressed constructive, i.e., giving enough information for realization, representations of processes. In particular, they represent intrinsic structures of computer programs. Hence, algorithm is an essence that is independent of how it happens to be represented and is similar to mathematical objects. Once the concept of algorithm is so rendered, its broader connotations virtually spell themselves out. As a result, algorithm appears as consisting of three components: *structure*, *representation* (linguistic, mechanical, electronic etc.), and *interpretation*.

It is important to understand that not all systems of rules represent algorithms. For example, you want to give a book to your friend Johns, who often comes to your office. So, you decide to take the book to your office (the first rule) and to give it to Johns when he comes to your office (the second rule). While these simple rules are fine for you, they are much too ambiguous for a computer. In order for a system of rules to be applicable to a computer, it must have certain characteristics. We specify these characteristics later in formal definitions of an algorithm. Now we only state that formalized functioning of complex systems (such as people) is mostly described and controlled by more general systems of rules than algorithmic structures. They are called procedures.

Definition 2.5. *A procedure is a compressed operational representation of a process in the form of a structure.*

For instance, you have a set of instructions for planting a garden where the first step instructed you to remove all large stones from the soil. This instruction may not be effective if there is a ten-ton rock buried just below ground level. So, this is not an algorithm, but only a procedure. However, if you have means to destroy this rock, this system of rules becomes an algorithm.

Definition 2.5 describes procedure in the theoretical sense because there is also a notion of procedure in the sense of programming. Namely, a *procedure in a program*, or *subroutine*, is a specifically organized sequence of instructions for performing a separate task. This allows the subroutine code to be called from multiple places of the program, even from within itself,