THESIS ON INFORMATICS AND SYSTEM ENGINEERING C128

Discrete Gravitational Swarm Optimization Algorithm for System Identification

MARGARITA SPITŠAKOVA

PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer Science on August 4, 2017.

Supervisor: Jaan Penjam, PhD
Department of Software Science
School of Information Technologies
Tallinn University of Technology
Tallinn, Estonia

Opponents: Per Kristian Lehre, PhD
School of Computer Science
University of Birmingham
Birmingham, United Kingdom

Daniil Chivilikhin, PhD
Computer Technologies Laboratory
ITMO University
Saint Peterburg, Russia

Defence of the thesis: September 7, 2017, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for doctoral or equivalent academic degree.

European Union
European Social Fund

Investing in your future

ISSN 1406–4731
ISBN 978–9949–83–143–2 (publication)
ISBN 978–9949–83–144–9 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C128

Diskreetne gravitatsioonilist vastasmõju arvestav osakeste parvega optimeerimise meetod süsteemide identifitseerimiseks

MARGARITA SPITŠAKOVA

TTÜ
KIRJASTUS

TABLE OF CONTENTS

Glossary

DFA	Deterministic FA.
$\mathrm{DFA}_{\emptyset}$	Unlabeled DFA.
DGSO	Discrete Gravitational Swarm Optimization.
FA	Finite Acceptor.
$\mathrm{FA}_{\emptyset}$	Unlabeled FA.
FSA	Finite State Acceptor.
FSM	Finite State Machine.
FST	Finite State Transducer.
GA	Genetic Algorithm.
GSA	Gravitational Search Algorithm.
ICDFA	Initially connected DFA.
ICFA	Initially connected FA.
MeFST	Mealy machine.
MoFST	Moore machine.
PSO	Particle Swarm Optimization.
SR(FST)	String representation of FST.
SR(MeFST)	String representation of Mealy machine.
SR(MoFST)	String representation of Moore machine.
$\mathrm{SR}^{\mathrm{B}}(\mathrm{FST})$	Binary string representation of FST.
$\mathrm{SR}^{\mathrm{B}}(\mathrm{MeFST})$	Binary string representation of Mealy machine.
$\mathrm{SR}^{\mathrm{B}}(\mathrm{MoFST})$	Binary string representation of Moore machine.
$\mathrm{SR}_{\mathrm{D}}(\mathrm{FST})$	String representation of FST with derived output function.
$\mathrm{SR}_{\mathrm{D}}(\mathrm{MeFST})$	String representation of Mealy machine with derived output function.
$\mathrm{SR}_{\mathrm{D}}(\mathrm{MoFST})$	String representation of Moore machine with derived output function.
$\mathrm{SR}_{\mathrm{S}}(\mathrm{FST})$	String representation of FST with separated output function.
$\mathrm{SR}_{\mathrm{S}}(\mathrm{MeFST})$	String representation of Mealy machine with separated output function.
$\mathrm{SR}_{\mathrm{S}}(\mathrm{MoFST})$	String representation of Moore machine with separated output function.

$\mathrm{cSR_D(FST)}$	Canonical string representation of FST with derived output function.
$\mathrm{cSR_S(FST)}$	Canonical string representation of FST.
$\mathrm{cSR_S(MeFST)}$	Canonical string representation of Mealy machine.
$\mathrm{cSR_S(MoFST)}$	Canonical string representation of Moore machine.

LIST OF PUBLICATIONS

A Spichakova, Margarita (2017). **Gravitationally Inspired Search Algorithm for Solving Agent Tasks.** Baltic Journal of Modern Computing, 5(1), 87 – 106.

B Spichakova, Margarita (2016). **Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions.** Proceedings of the Estonian Academy of Sciences, 65(1), 15 – 27.

C Spichakova, Margarita (2013). **An approach to inference of finite state machines based on a gravitationally-inspired search algorithm.** Proceedings of the Estonian Academy of Sciences, 62(1), 39 – 46.

D Spichakova, Margarita (2011). **An approach to inference of finite state machines based on gravitationally-inspired search algorithm.** 12th Symposium on Programming Languages and Software Tools, SPLST'11: Tallinn, Estonia, 5–7 October 2011, Proceedings. Tallinn: TUT Press, 185 – 195.

OTHER PUBLICATIONS

E Ojamaa, Andres; Kotkas, Vahur; **Spichakova, Margarita**; Penjam, Jaan (2013). **Developing a lean mass customization based manufacturing.** 2013 IEEE 16th International Conference on Computational Science and Engineering, CSE 2013: Sydney, Australia, 3–5 December, 2013, Proceedings. Piscataway, NJ: IEEE, 28 – 33.

AUTHOR'S CONTRIBUTIONS TO THE PUBLICATIONS

The numeration of the list corresponds to the numeration in the list of publications. The candidate is the only author of all the listed publications. Contribution to the papers in this thesis are:

A	The article covers the application of the method for inference of Mealy machine, based on *Gravitationally Inspired Search Algorithm* to *artificial ant problem*.
B	The article describes the variation of a heuristic search algorithm: *Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions* based on the hybridization of *Particle Swarm Optimization* and *Gravitational Search Algorithm*.
C	The article describes the method for inference of Moore machine based on a *binary Gravitational Search Algorithm* with an application for system identification.
D	The article is a conference paper, which was published as Publication C being revised.

ACCOMPANYING CODE

The Java code, which implements the methods proposed in the thesis, is available at `https://github.com/dragazhar/s-ma-u-g`.

The source code, which covers the implementation and the experimental part of Publication B is available at `https://github.com/dragazhar/dioGIpso`.

INTRODUCTION

The *identification task* (or a similar reverse engineering task) is an issue related to constructing a model from examples of system behavior or specification or by simulating the system. For instance, the task of hardware reverse engineering consists of constructing a model, e.g. a *finite state machine* that mimics the behavior of an integral circuit [1]. A model is reconstructed from the observable data, i.e. inputs and outputs of the integral circuit.

Finite state machines (FSMs) present a class of abstract models, which can be characterized by a finite number of states and transitions between them, which are triggered by event. These abstract models are applicable to a wide set of problems, such as speech recognition, hardware modeling, software modeling, artificial intelligence, etc. FSM models are very useful for system identification, as many systems have FSM-like behavior.

Problem-specific deterministic procedure is used for solving several of the system identification tasks. For example, the problem of grammatical inference is solvable by a deterministic algorithm that constructs a prefix-tree acceptor and applies state merging. However, unfortunately in most cases either no solutions exist for the problem or the existing solutions are too complex. For example, it is proven that the grammatical inference problem is NP-complete [2]. To solve the system identification problem in the case of the non-polynomial complexity another method must be found.

Heuristics is a technique, which sometimes allows to solve problems faster and more easily than classical methods.

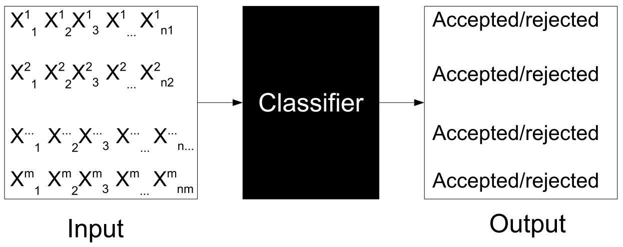
Thus, heuristics can be applied to solve the identification task in the case of the non-polynomial complexity. There are two approaches – firstly, the direct one, whereby heuristic optimization is applied directly to the search problem without any recourse to classical solutions. Secondly, an indirect approach can be taken whereby a heuristic algorithm is only additional to the classical one. As we are looking for a general algorithm for constructing FSM which is applicable to a wide range of problems (there may be no existing deterministic procedures), we use the direct approach. There are many different heuristic optimization techniques, but here we propose to apply population-based search algorithms, such as *Particle Swarm Optimization* or *Genetic Algorithms*.

Motivation and existing solutions

In this section we describe the set of tasks, that are quite common in the field of FSM inference and can be used for benchmarking and search algorithm analysis. Some of the tasks are discussed later in the experimental part (see Chapter 4).

Grammatical inference

The term *grammatical inference* usually describes the set of problems, which require inference of one of the structures for recognizing languages, e.g. automaton, grammar, regular expression (more details in Section 2.1). More generally, the *language recognition task* can be seen as building up a classifier system, which separates all words build over alphabet into two sets – accepted and rejected strings (see Figure 1). Thus, the classifier returns a classification for each input string.



$X^1_1\ X^1_2 X^1_3\ X^1_{\dots}\ X^1_{n1}$
$X^2_1\ X^2_2 X^2_3\ X^2_{\dots}\ X^2_{n2}$
$X^{\dots}_1\ X^{\dots}_2 X^{\dots}_3\ X^{\dots}_{\dots}\ X^{\dots}_{n\dots}$
$X^m_1\ X^m_2 X^m_3\ X^m_{\dots}\ X^m_{nm}$
Classifier
Accepted/rejected
Accepted/rejected
Accepted/rejected
Accepted/rejected
Input
Output

Figure 1 Classifier

The inference problem in this case lies in constructing the model using only a given set of classified strings (a training set usually consists of positive and negative examples). In general, the grammatical inference task is an NP- complete problem [2]. However, there are several deterministic methods (building up a Prefix-tree acceptor, evidence-driven state merging, etc.). Nevertheless, using heuristic methods can be very effective for grammatical inference.

The choice of structure, which describes the language that would be inferred, is based on the language type. *Finite acceptors*, regular grammar or regular expressions can be used for regular languages. Other classes of languages (e.g. context-free languages) require more complex structures, such as context-free grammars or push-down automata. The heuristic inference method is also successfully utilized for solving the identification task for non-regular languages.

Existing solutions

Hingston [3] shows how a *Genetic Algorithm* (GA) can be used for the inference of a regular language from a set of positive and optionally also negative examples. That method is based on constructing a prefix-tree acceptor and on state-merging using a GA. The method is benchmarked on a set of Tomita regular languages [4] and the results are compared to RPNI (Regular Positive Negative Inference).

Kohli [5] proposed a method for *finite acceptor* (FA) inference from positive and negative examples (the training set must be structurally complete), based on a *Genetic Algorithm*, wherein the FA is represented by a two- dimensional chromosome and all GA operations are defined in tabular form. The method was tested on 7 regular languages and compared with GIG, RPNI methods.

Horihan and Lu [6] introduced a method for the optimization of incremental evolution during the construction of the FA, which recognizes the strings represented by regular expressions, based on *Particle Swarm Optimization*, wherein the FA is presented by a transition graph and a progressive fitness function.

Lucas and Reynolds [7] proposed an evolutionary method for learning deterministic FA (DFA) that evolves only in the transition matrix and uses a simple deterministic procedure to optimally assign state labels. This method was compared to Evidence Driven State Merging on random data sets.

Bongard and Lipson [8] described an active learning approach to the problem of *grammatical inference* (estimation-exploration algorithm) and presented a comparison with evolutionary methods and *Evidence Driven State Merging*.

Lankhorst [9] proposed a GA for the inference of context-free grammars, Lucas [10] presented chromosome design based on normal form for the evolution of context-free grammars. Zomorodian [11] applied *Genetic Programming* to the construction of push-down automata for learning context-free languages. Lankhorst [12] continued previous work [31] and presented a GA for learning non-deterministic push-down automata.

System identification

The *identification task* is more complex than the *classification task*. The problem lies in constructing a model that mimics the behavior of a system (i.e. a black box). The system has more complex behavior – for each input string (sequence of input signals) it returns the output string (sequence of output signals) (see Figure 2). Such behavior can be simulated by a *finite state transducer* (FST) (see Subsection 2.1.3).

Therefore, the system identification task, i.e. constructing a model based on behavior can be described as a task of FST inference from the set of input/output pairs. Such models are useful for representing hardware devices (for example in reverse engineering task) and for language processing (translating between two languages). The system identification problem is discussed in more detail in Section 4.1.

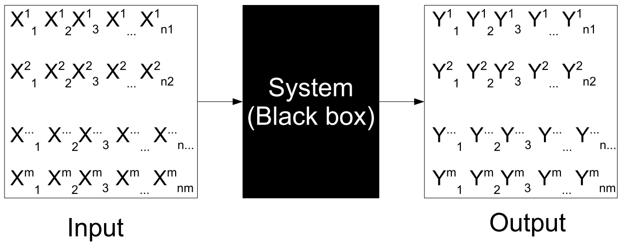

$X^1_1\ X^1_2 X^1_3\ X^1_{\dots}\ X^1_{n1}$
$X^2_1\ X^2_2 X^2_3\ X^2_{\dots}\ X^2_{n2}$
$X^{\dots}_1\ X^{\dots}_2 X^{\dots}_3\ X^{\dots}_{\dots}\ X^{\dots}_{n\dots}$
$X^m_1\ X^m_2 X^m_3\ X^m_{\dots}\ X^m_{nm}$
Input
System (Black box)
$Y^1_1\ Y^1_2 Y^1_3\ Y^1_{\dots}\ Y^1_{n1}$
$Y^2_1\ Y^2_2 Y^2_3\ Y^2_{\dots}\ Y^2_{n2}$
$Y^{\dots}_1\ Y^{\dots}_2 Y^{\dots}_3\ Y^{\dots}_{\dots}\ Y^{\dots}_{n\dots}$
$Y^m_1\ Y^m_2 Y^m_3\ Y^m_{\dots}\ Y^m_{nm}$
Output

Figure 2 System as a black box

Existing solutions

Manovit, Aporntewan and Chongstitvatana [13] presented the *Genetic Algorithm* approach wherein each individual was presented by bit string to synthesize a synchronous sequential logic circuit, which can be described by FSM, from a partial input/output sequence (circuit specification). This method was tested on reconstructing ’Frequency Divider’, ’Odd Parity Detector’, ’Modulo-5 Detector’, ’Serial Adder’.

Chongstitvatana and Aporntewan [14] continued their work and presented the improvement method, which increases the correctness percentage of the finite state machine FSM synthesis using multiple partial input/output sequences.

Ngom, Baron and Geffroy [15] presented a new approach based on genetic simulation for Moore machine identification, where the Moore machine is presented by a string of integers. Tongchim and Chongstitvatana [16] presented the parallel implementation of Genetic Algorithm for FSM inference. Niparnan and Chongstitvatana [17] suggested using the *Genetic Algorithm* only for inferring the transition function of a Mealy machine and using the deterministic procedure for constructing output function. Geng [18] presented the GA method for the identification of asynchronous FSM. In this method both one-dimensional and two-dimensional representation are discussed.

Shayani and Bentley [19] presented a very specific biologically inspired boolean representation of FSM (input, current state, next state) and a GA-based method for inferring the output function of a Moore machine from I/O pairs. Naidoo and Pillay [20] presented the method for the inference of Mealy machines that is based on *Genetic Programming* (represented by a deterministic graph).

Artificial ant

The *artificial ant* task was initially proposed by Jefferson [21] in 1991 to benchmark Evolutionary Algorithms. The task consists of designing a *trail tracker*, which acts as an artificial ant and follows the trail that contains food (see Figure 3). The goal of the ant is to collect the maximum amount of food for a limited number of steps (traditionally 200 steps).

The ant is constructed as a FSM (Mealy type) like *agent*. The input of such a machine is only one variable – whether there is food in the next cell, with values ’Food’ and ’Empty’. The outputs are defined as the actions of the ant: ’Wait’, ’Turn left’, ’Turn right’, ’Move’.

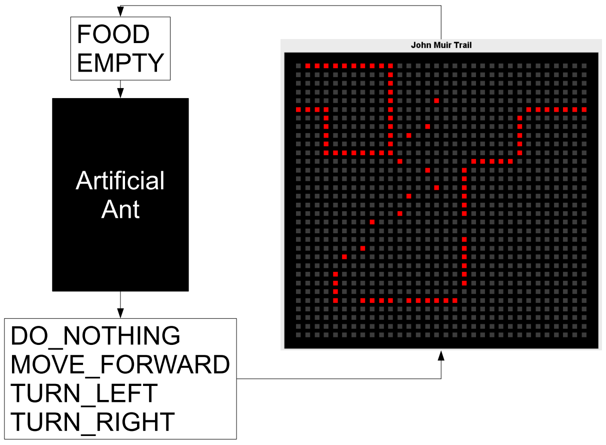


Figure 3 Artificial ant

Existing solutions

The *artificial ant* task has been researched by many authors [22], [23], [24], [25] and is often used for benchmarking the Evolutionary Algorithms. Therefore, we also use it as the benchmark test for our method (see Section 4.2).

There are several methods for solving the *artificial ant* task, some authors use FSM inference [22], [23], [25], others apply *Genetic Programming* [22]. For example, Chellapilla [26] used the modular Mealy machine as the artificial ant and the evolutionary programming procedure as the optimization algorithm. It is also possible to solve the *artificial ant* task by artificial neural network learning [21].

Other possible applications

There is a set of other applications that can be solved by FSM identification:

1. prediction of binary sequences [27], [28],
2. game theory (modeling agents and strategies), for example, strategies for protecting resources [29], prisoner’s dilemma [27], negotiations strategies [30],
3. image processing tasks [31], such as automatic target detection [32], image chain code (code-based recognition of binary images) [33], [31],

4. modeling controllers [34], e.g. robot controllers [35], pulse generator, counters, control water pump [34], thermostat [27], etc.

Problem statement

Although there are several tasks from different areas, which can be solved by the identification of FSM, these have a number of common issues:

- As a result of the modeling/identification process, we want to obtain a FSM (Mealy machine, Moore machine, FA, etc.).
- The task defines the type of the machine, its alphabets and number of states. Using task description, we can define the measurement that shows how a given FSM solves the task.
- The search process requires stochastic optimization algorithms (*Evolutionary Algorithm*, *Genetic Algorithm*, etc.).

We define our *general task* as **building a problem-independent population-based heuristic optimization technique, which is applicable to the task of constructing a finite state machine that models the behavior of the system, from examples of the behavior of the system or its specification.** To reach that *goal* we need to answer several *questions*:

Question I. Which problems can be solved by inference of FSMs?

Question II. What do these problems have in common?

Question III. Is it possible to build up a unified method, which solves several of these problems?

Question IV. Why do we need to choose stochastic optimization?

Question V. Which stochastic algorithm can we use in this case?

Question VI. How to adapt existing stochastic algorithms for FSM identification?

Question VII. Is it possible to minimize search space?

Question VIII. Is our proposed algorithm better than other existing algorithms?

CONTRIBUTIONS OF THE THESIS

To answer the questions listed in the introduction, we propose the following methods for the inference of FSMs.

I. Modular system

We propose the **modular system**, which helps us to answer Questions I–III. Although most of the existing algorithms are made for solving an exact problem, we discover that those problems have several characteristics in common and can be solved by a unified method. Hence, we propose to separate the problem statement from the solving algorithm (see Chapter 1). This allows us:

- to use the same search algorithm for different problems or
- to use different search algorithms for one problem.

The proposed approach was applied to several well-known benchmarks, such as *artificial ant*, the system identification problem and the binary sequence predictor (see Chapter 4). The experiments showed that such varied problems can be solved by one unified method.

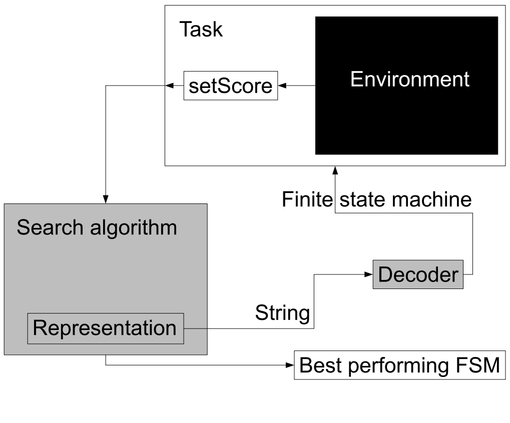


Figure 4 The outline of the FST search process

The proposed system which implements the ideas of unified methods for FSM identification consists of three main modules (see Figure 4). Each system module presents an independent part of the system and can be replaced by another implementation or definition:

- *'Task' module* includes the implementation of basic concepts and definitions that are used to describe the problem statement. In order to describe the problem, we need to define the type of FSM, choose alphabets, the number of states and construct the FSM evaluation algorithm. These parameters can be derived from the problem specification.

- *'Search algorithm' module* includes the implementation of different stochastic optimization algorithms, such as the *Genetic Algorithm* or *Particle Swarm Optimization*.

- *'Representation+Decoder' module* contains algorithms for defining FSM and the string representation of FSM.

II. Search space representation technique

The efficiency of the search method depends not only on the choice of search algorithm, but also on the structure and size of the search space. There are several techniques used for the representation of FSM in the context of stochastic optimization (see Chapter 2).

The most popular heuristic search algorithms, such as *Evolutionary Algorithm* or *Particle Swarm Optimization*, work with string representations, while graph representation requires a special search algorithm, e.g. *Genetic Programming*. Therefore, string representation was chosen for our purposes.

First of all, we propose string representation, which handles the FSM transition and output functions separately (see Sections 2.3 and 2.4). Secondly, we discuss some problems common for string representations, namely with isomorphisms and FSMs with unreachable states. Finally, we adapt the canonical string representation method (Section 2.7), which is initially used for the enumeration of finite acceptors.

The new search space representation (Section 2.8)

- helps to reduce the search space size (Question VII),

- divides the search space into non-intersecting parts, which can be handled separately (Section 3.2),

- introduces one-to-one correspondence between FSM and triple of numbers, which can be used for hashing (Section 3.2.3).

III. Two–stage search algorithm

In the context of FSM identification, we propose a new two-stage search algorithm (see Chapter 3).

Usually, the search algorithm considers the entire search space, but due to the specific string representation of FSM, the search space consists of several non-intersecting subsets. Thus, we propose to subdivide the search algorithm into two phases:

1. the search space is subdivided to subspaces. A score value is assigned for each subspace and the subspace with the best score value is returned,
2. a more detailed search in the chosen subspace is carried out, and if a solution is not found, the next subspace with a higher score is searched.

IV. Discrete gravitational swarm optimization algorithm

Deterministic methods exist for some of the system identification problems, although they cannot always be applied to systems with complex behavior due to the growing space and time complexity. Therefore, the stochastic optimization is very helpful for the identification of such complex systems. The family of stochastic optimization methods is big and constantly developing. Traditionally, the different types of the evolutionary computation methods are used for the task of FSM identification, but there are newer methods based on social behavior algorithms (Questions IV–VI).

Currently, one of the most popular stochastic optimization methods is *Particle Swarm Optimization* (PSO), which is inspired by the social behavior of a set of objects, e.g. a flock of birds or a school of fish [36]. Initially, a PSO is designed for the real-valued vector that is uncomfortable for the representation of FSM, as state machines are traditionally represented by transition graphs, or a transition table (or their modifications). Representation is further elaborated upon in Chapter 2.

Consequently, we propose the following modifications:

- Discrete variation of the optimization algorithm. We propose to modify the PSO search operators to be able to explore the discrete search space.
- Gravitational interactions between particles. Although PSO works well on smooth search spaces, we need to add more sophisticated behavior to our search algorithm, because the search space in our case has a more complex landscape. Thus, we propose to adapt the *Gravitational Search Algorithm* (GSA) and present the new modified PSO algorithm based on gravitational field interactions.

The proposed improvements help to create a new heuristic search algorithm, which is applicable to a discrete search space and has fewer control parameters, which makes adjusting the algorithm simpler (see Publication B).

OUTLINE OF THE THESIS

This thesis is structured as follows: Chapter 1 describes the set of problems and general ideas concerning the identification of FSM. The modular system is introduced. Chapter 2 gives an overview of the existing string representations of FSM and presents the adaptation of the *canonical string representation* method to the problem of search space representation. Chapter 3 contains the basic definitions of existing stochastic optimization methods and introduces a new search algorithm. Chapter 4 covers the results of applying the proposed method to benchmark tasks such as *system identification*, *artificial ant problem* and *binary string predictor*.

1. IDENTIFICATION OF STATE MACHINES

Identification is an *inference* process, which deduces an internal representation of a system (*internal model*) from samples of its functioning (*external model*) [37]. Inference of finite state machines (FSMs) is widely applied in different fields, such as logical design, verification and software systems, for example *grammatical inference*, *system identification*, *artificial ant* problem, agent and controllers modeling, image processing, etc. Although these tasks are from different areas, they have several common issues that can be used to create a generalized problem solver:

- As a result of the modeling/identification process we want to obtain a FSM, such as Mealy machine, Moore machine, FA etc.
- The task defines the machine type, its alphabets and number of states. We can define the measurement that shows how a given FSM solves the task by defining the evaluation function from task description.
- The search process requires a stochastic optimization algorithm (e.g. *Evolutionary Algorithm*, *Genetic Algorithm*).

Using these common ideas we can construct the system, which helps to formalize the problem statement and uses search algorithms for solving the FSM identification problem.

1.1. Modular System for FSM Identification

We propose a modular system (see Figure 1.1), wherein each component is independent and can be replaced by other implementations. The **'Task'** module includes definitions and implementation required for defining the problem statement, such as the FSM type, alphabets, its number of states, and the FSM evaluation process.The **'Search Algorithm'** module contains the implementation of one of the the stochastic optimization algorithms, for example the *Genetic Algorithm* or *Particle Swarm Optimization*. The **'Representation+Decoder'** module implements the algorithms for FSM definition and its string representation.

The proposed system allows to either use the same algorithms for different problems or to try different algorithms for one problem. On the other hand,

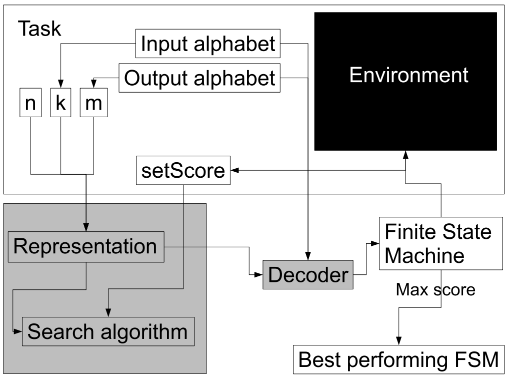


Figure 1.1 Inference of finite state machines

the problem-specific approach assumes the specialized algorithms and additional knowledge that can be extracted from the problem statement.

The implementation consists of several packages (Figure 1.2), each of which includes the implementation of a certain part of the method:

- `FSM` package presents the basic functionality required for processing the FSM structure and implements the main definitions (see Section 2.1).
- `Representation` package contains the implementation of methods required for constructing, generating and decoding processes for different string representations of the FSM (see Chapter 2).
- `Search Algorithm` package contains the functionality, which defines and implements the whole search process (see Chapter 3).
- `Problems` package defines the problems and contains implementation for each test problem used for benchmarking (see Chapters 4).
- `Visualization` package contains entities required for visualizing the search process (discussed in Subsection 3.2.7).

These components must be considered in more details. To begin with, there are three main modules – **’Task’** module, **’Search algorithm’** module and **’Representation+Decoder’** module.

1.1.1. ’Task’ module

Before starting the search process we need to formulate the problem statement. The **’Task’** module must contain the formalized description of the problem and describe the behavior of the system to be modeled.

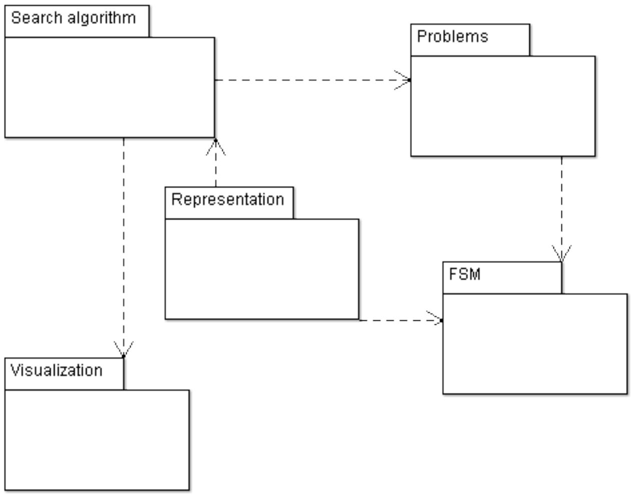


Figure 1.2 General structure

This module consists of three parts:

- The *environment* is a formalization of the system description. In essence, it is the training data of another system behavior description that can be used for simulation.
- The *agent* is a formal description of FSM behavior, which defines the machine type and corresponding work-flow. The *alphabets*, required for FSM work-flow, are observable from the *Environment* description. The agent is closer to the idea of the model, which must be learned to act in the environment.
- The *score* function assigns the evaluation function value for each FSM. It must return value in the range $[0 \dots 1]$, with value '1.0' being optimal. It shows how well the agent acts in a given environment, i.e. how well the model describes the system.

1.1.2. 'Representation+Decoder' module

There are several traditional representations of FSM – transition table, transition graph, matrices, etc. The choice of representation type depends on the search algorithm type. For some algorithms, it is better to choose graph structures, some of the algorithms work on strings, others only on binary strings. Each FSM representation must allow for correct transformations of $FSM \rightarrow Representation(FSM)$ and vice versa, $Representation(FSM) \rightarrow$

FSM. Random generation of points in search space and operations on the points that store their properties and correctness must also be possible.

In our approach, decimal string representation of FSM is used, which is in fact constructed as a concatenation of transition table rows. The **'Decoder'** module contains the algorithms required for transformation of $Representation(FSM) \to FSM$. The information necessary for a correct work process is the type of machine, the number of states n, the input alphabet Σ and its size k, the output alphabet Δ and its size m.

This representation was chosen to eliminate the connection between the FSM type and search algorithm construction. For each FSM type, there is a transformation algorithm, which is part of the **'Decoder'** module. Thus, the search algorithm works on the abstract string level. To evaluate the FSM, we need to transform string to FSM and pass it to the **'Task'** module for evaluation.

1.1.3. 'Search algorithm' module

According to the problem description, we need to construct the search algorithm, which works on the abstract string level. The search algorithm knows nothing about FSMs and problem description except the set of strings that represents FSM and the evaluation value for a given string, which is computed by the **'Task'** module. The main goal of this algorithm to choose the best string, which in fact represents the best FSM. In our approach, the stochastic optimization algorithm is employed.

Population-based meta-heuristic search algorithms, such as *Evolutionary Algorithm*, *Particle Swarm Optimization*, *Simulated Annealing*, etc., have some characteristics in common. Those properties allow us to unify the problem statement in order to apply different optimization techniques and choose the optimal one:

- The *search space* is defined as a set of points, where each point represents one *candidate solution*. Usually, the solutions are presented not directly, but by some other structures that are comfortable for modification algorithms. Initially, a certain fixed amount of points is generated randomly. For instance, this set of points is known as *population* in *Evolutionary Algorithms* or *swarm* in *Particle Swarm Optimization*.

- The function we need to optimize is known as the *evaluation function*. Using this function, we can assign the score value for each point in search space that shows the usefulness of this solution. The optimization task is to find a minimum or maximum, depending on the definition of the function. For example, for Evolutionary Algorithms this function is known as the *fitness function*.

- The search algorithm contains modification algorithms for constructing a new solution from existing ones, for example *selection*, *mutation* and *crossover* for *Evolutionary Algorithms*, *move* for *Particle Swarm Optimization*. Those modification algorithms have two main aims – exploration (the ability to search the whole search space) and exploitation (the ability to find a better solution in a local situation).

The population-based meta-heuristic optimization technique can be described as a consequent application of a modification algorithm to an initial set of solutions until the optimal solution with the best evaluation function value is found.

1.2. Modular System Work-flow

The FSM identification process (Figure 1.1) can be described as follows:

1. The task must be defined (this is done by defining the *Task* module). Using this data we can construct input and output alphabets, choose the machine type and implement the *score* function.
2. The parameters and type of search algorithm is chosen. One of the parameters is the number of states n.
3. The search process consists of four phases. Firstly, the search algorithm chooses one string which is decoded into FSM. This FSM is then evaluated and lastly, the string with the score value is returned to the search algorithm.

1.3. Examples of Modular System Applications

This section highlights several examples of task formalization. The modules **'Representation+Decoder'** and **'Search algorithm'** are the same for all tasks. However, the **'Task'** module varies depending on the problem at hand.

1.3.1. Grammatical inference

It is useful to examine the problem of *grammatical inference* in more detail. In order to formalize this problem, we need to define the following three formalisms:

- the *agent* is a finite state acceptor (FSA) with the input alphabet observed from the set of examples,
- the *environment* is a set of positive (accepted by FSA) and sometimes negative (rejected) examples, i.e. a set of strings,
- the *score functions* shows how well the constructed FSA describes a training set, which is usually constructed on the basis of string difference functions.

1.3.2. Artificial ant problem

In the *artificial ant* problem, the formalisms are the following:

- the *agent* is a Mealy machine with the input alphabet $\Sigma = \{FOOD, NO-FOOD\}$ and the output alphabet $\Delta = \{LEFT, WAIT, RIGHT, MOVE\}$, which simulates the behavior of the ant,
- the *environment* is a grid with pre-given locations for the food, and for where the ant is moving
- the *score function* can be defined in several ways. One possible definition [23] is that the score function shows how many food cells were visited within 200 steps. To define score function in the range $[0 \dots 1]$] we can divide the number of visited food cells by the total number of cells with food.

1.3.3. System identification

The *System identification* problem can be deconstructed as follows:

- the *agent* is a finite state transducer (a Mealy or Moore machine) with input and output alphabets that can be observed form the training set. The type of FST is user defined,
- the *environment* is given by the training set that contains the input and output pairs, which are observed from the system we need to model,
- the *score function* shows how the well the constructed FST describes the given training set (data consistency) and usually the *score function* is constructed using the string distance functions.

1.4. Conclusion

The identification of FSMs is a classical task that is encountered in a wide range of applications. In some situations, there is a deterministic procedure for solving the task. However, in certain cases, we need to apply other search algorithms, e.g. stochastic optimization. This chapter gives a brief overview of the system, which allows to solve the FSM identification task for different problem statements without significant changes in the search process. The proposed system consists of modules, each of which presents a subsystem that is responsible for specific algorithms — one of the modules is for the formalization of the problem statement, the second one is for the representation of FSM as string and the third one implements the search algorithm. As a result, there is no need to alter the entire system, if the problem statement is changed. We demonstrated how well-known problems can be formalized for such system.

2. STRING REPRESENTATION OF FINITE STATE MACHINES

The term finite state machine (FSM) describes a class of models that are characterized by having a finite number of states. The class of FSMs can be subdivided into several subclasses, the most important of which are finite acceptor (FA), finite state machine without output, or finite automaton, and the finite state transducer (FST), a finite state machine with output. The class of FSTs includes several modifications of FST, but this research focuses on the Moore and Mealy machine.

2.1. Preliminaries. Finite State Machines

This section provides a brief overview of the FSM theory, including basic definitions and algorithms.

2.1.1. Alphabet, words, language

A symbol is a basic component of strings and alphabets.

Definition 2.1 (Alphabet) *The alphabet Σ is a set of symbols $\Sigma = \{a_1, \ldots, a_n\}$.*

Definition 2.2 (String(Word)) *A sequence of symbols from alphabet Σ is called string(word). The empty string is denoted as ϵ.*

Definition 2.3 (Length of word) *If w is a string then $|w|$ denotes the number of symbols in w and is called the length of w, $|\epsilon| = 0$.*

Definition 2.4 (Equal words) *Two strings w and u are equal if they contain the same number of symbols $|w| == |u|$ in the same order.*

Definition 2.5 (Concatenation) *Given two strings w, $u \in \Sigma^*$, we can form a new string $w \cdot u$ ($w \cdot u = wu$), called the concatenation of w and u. Concatenation of w and u means adjoining the symbols in u to symbols in w. The order in which strings are concatenated is important. The concatenation with an empty string ϵ has the following property:*

$$\epsilon w = w\epsilon = w$$

Example 2.1 *Let's define $x = a_1a_2\ldots a_n$ and $y = b_1b_2\ldots b_m$, then $x \cdot y = a_1a_2\ldots a_nb_1b_2\ldots b_m$*

Definition 2.6 (Prefix, suffix) *Let w,$u \in \Sigma^*$. If $x = wu$, then w is called the prefix of x and u is called suffix of x.*

The set of all strings over alphabet Σ is denoted by Σ^*.

Definition 2.7 (Language) *For any alphabet Σ, a subset of Σ^* is called language.*

Definition 2.8 (Operations on languages) *If L and M are languages over alphabet Σ, then:*

- *$L \cup M$ is the union of L and M,*
- *$L \cap M$ is the intersection of L and M,*
- *$\bar{L}$ is a complement of L,*
- *$L \cdot M = \{wu : w \in L \text{ and } u \in M\}$ is the product of L and M. A string belongs to LM, if it can be written as a string in L concatenated with a string in M,*
- *$L^0 = \{\epsilon\}$ and $L^{n+1} = L^n \cdot L$. For $n > 0$, the language L^n consists of all strings w of form $w = u_1u\ldots u_n$ where $u_i \in L$,*
- *$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$ is the Kleene Star of language L.*

Measuring distances between words

We specify several functions for computing the distance between strings. First of all, we define the distance between symbols.

Definition 2.9 (Distance between symbols $\Delta(a, b)$) *We specify a function $\Delta(a, b)$, where a, b are symbols in a certain alphabet:*

$$\Delta(a,b) = \begin{cases} 0 & : a == b \\ 1 & : a \neq b \end{cases} \tag{2.1}$$

that returns 1 if symbols are not equal.

Based on $\Delta(a, b)$ function (Definition 2.9) we can specify several functions for measuring the distances or similarities between strings.

Definition 2.10 (Strict distance) *is defined as:*

$$D_{strict}(x,y) = \begin{cases} 0 & : x == y \\ 1 & : x \neq y \end{cases} \tag{2.2}$$

The strict distance returns 0, if the strings are equal (see Definition 2.4), otherwise it returns 1.

The computing of strict distance has the worst case time complexity $\mathcal{O}(|x|)$ (when $|x| == |y|$), otherwise (if $|x| \neq |y|$, then function returns 1) it is constant in time .

Definition 2.11 (Hamming distance) *is defined as the following:*

$$D_{Ham}(x, y) = \Sigma_{i=1}^{Min(|x|,|y|)} \Delta(x_i, y_i) \tag{2.3}$$

Hamming distance returns the number of differences (different symbols). Computing Hamming distance has a complexity of $\mathcal{O}(Min(|x|, |y|))$.

Similarly to Definition 2.11 we can define *Hamming similarity*, which is sometimes more applicable as the string distance function.

Definition 2.12 (Hamming similarity) *is defined as:*

$$S_{Ham}(x, y) = Min(|x|, |y|) - \Sigma_{i=1}^{Min(|x|,|y|)} \Delta(x_i, y_i) \tag{2.4}$$

or

$$S_{Ham}(x, y) = \Sigma_{i=1}^{Min(|x|,|y|)} \overline{\Delta}(x_i, y_i) \tag{2.5}$$

Hamming similarity returns the number of equal symbols.

Definition 2.13 (Length of maximal equal prefix) *is defined as:*

$$D_{LP}(x, y) = \Sigma_{i=1}^{x=y} \Delta(x_i, y_i) \tag{2.6}$$

The analysis of strings is stopped, when the first difference between symbols is found.

Example 2.2 *The comparison between string distance functions $D_{LP}(x, y)$ (Definition 2.13), D_{Ham} (Definition 2.11), D_{strict} (Definition 2.10) is shown on the string distance between "qwerty" and the strings in each column (see Table 2.1).*

Table 2.1 String distance functions

Function	"qwerty"	"qeerty"	"qweryt"
D_{strict}	0	1	1
D_{Ham}	0	1	2
D_{LP}	6	1	4

2.1.2. Finite acceptor

In this subsection we introduce the basic knowledge of FA, which is a special case of FSM. The FSM takes a string in a specific alphabet Σ and outputs the value 'yes' if the string is accepted by machine or 'no' if the string is rejected. The set of all possible input strings can be divided into two classes: accepted strings and rejected strings. The FA itself can be viewed as the classifier (see Figure 2.1).

$s(t)...s(i)...s(2)s(1)$
Finite Acceptor
Output signal

Figure 2.1 Conception of finite acceptor

Definition 2.14 (Finite acceptor (FA)) *is a five-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite input alphabet,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of final states,*
- δ *is a transition function:* $\delta : Q \times \Sigma \to Q$*.*

The work-flow of the model is presented in Algorithm 1.

Definition 2.15 (Deterministic finite acceptor (DFA)) *If for each state* $q \in Q$ *and each symbol* $a \in \Sigma$*, there exists at most one transition (i.e.,* $|\delta(q, a)| \leq 1$*), the acceptor is a deterministic finite acceptor (DFA), otherwise it is a non-deterministic finite acceptor (NFA).*

Theorem 2.1 *Let* L *be a language accepted by a non-deterministic finite automaton, then there exists a deterministic finite automaton that accepts L [38].*

There are two popular ways to represent FA – a transition diagram (graph) and a transition table.

Definition 2.16 (FA Transition diagram) *A transition graph is a special case of the directed labeled graph, where vertices are labeled by Q; there is an arrow labeled* 'a' *from vertex labeled* 's' *to vertex labeled* 't' *exactly when* $t \in \sigma(s, a)$*. The initial state is marked by an inward-pointing arrow and final state by double circles.*

Definition 2.17 (Yield operator) *If* $M = (Q, \Sigma, \delta, q_0, F)$ *is* FA*,* $a \in \Sigma \cup \epsilon$*, when we say* $(q, aw) \vdash_M (p, w)$*, iff* $p \in \delta(q, a)$*, where* $\vdash_M$ *is called yield operator.*

Algorithm 1 FA work-flow

```
1: function runMachine($inputString$)
2:     $q_c \leftarrow q_0$
3:     for $i = 0 \rightarrow$ length$(inputString) - 1$ do
4:         makeTransition($inputString[i]$)
5:     end for
6:     if $q_c \in F$ then
7:         $output \leftarrow accepted$
8:     else
9:         $output \leftarrow rejected$
10:    end if
11: end function

12: function makeTransition($ic$)
13:     $transition \leftarrow TransitionTable[q_c.label][\Sigma.position(ic)]$
14:     $q_c \leftarrow q_{transition.toState}$
15: end function
```

Definition 2.18 (FA Transition table) *A transition table is a special representation method for* FA*, where each row presents states from Q and each column corresponds to an input symbol from Σ. Each cell contains $\delta(q, \sigma)$ with respect to column and row numbers.*

Example 2.3 *An example of a simple* FA *is represented as the transition diagram (see Figure 2.2) and as the transition table (see Figure 2.3). The five components of the* FA *are:*

- *the set of states $\{q_0, f_1\}$,*
- *the input alphabet $\Sigma = \{a, b\}$,*
- *the initial state q_0,*
- *the set of final states $F = \{f_1\}$,*
- *the transition function $\delta : Q \times \Sigma \rightarrow Q$ is defined as*

$$\delta(q_0, a) = q_0,\ \delta(q_0, b) = f_1,\ \delta(f_1, a) = q_0,\ \delta(f_1, b) = f_1$$

Definition 2.19 (String accepted by an acceptor) *A string $'w'$ is said to be accepted by an acceptor M iff $(q_0, w) \vdash_M^* (p, \epsilon)$ for some $p \in F$, i.e. there exists a finite sequence of transitions, corresponding to the input string w, from the initial state to a certain final state.*

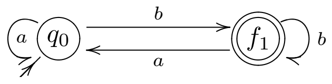


Figure 2.2 Example of a transition diagram for FA.

	a	b
$\rightarrow q_0$	q_0	f_1
$\leftarrow f_1$	q_0	f_1

Figure 2.3 Example of a transition table for FA

Definition 2.20 (Language accepted by an acceptor) *The language accepted by M, is denoted as $L(M)$ and defined as:*

$$L(M) = \{w \mid \exists p \in F : (q_0, w) \vdash_M^* (p, \epsilon)\}.$$

Definition 2.21 (Accessible state) *Let $A = (Q, \Sigma, i, q_0, F)$ be a* FA. *We say that state $q \in Q$ is accessible if there is a string $x \in \Sigma^*$ so that $q_0 \cdot x = q$, where $q_0 \cdot x = q$ means that state q can be reached from state q_0 by making transitions according to a corresponding symbol $x_0 \ldots x_n$.*

$$q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} \ldots \xrightarrow{x_n} q$$

A state that is not accessible is called inaccessible.

Definition 2.22 (Unlabeled FA ($FA_\emptyset$)) *The structure* FA$_\emptyset$ =(Q, Σ, δ, q_0) *denotes* FA *without empty states.*

Example 2.4 *Suppose we have* DFA$_\emptyset$ *with 4 states (see Figure 2.4). State labeled '3' is unreachable from the initial state, so it is redundant and can be removed without any damage to the work cycle. This leads us to the* FSM *with only three states.*

Definition 2.23 (Accessible FA. Initially connected FA (ICFA)) *An acceptor* FA *is accessible (or initially connected) if its every state is accessible.*

Definition 2.24 (FA Isomorphism) *Two* DFA $A_1 = (Q^1, \Sigma^1, \delta^1, q_0^1, F^1)$ *and* $A_2 = (Q^2, \Sigma^2, \delta^2, q_0^2, F^2)$ *are called isomorphic by states if* $|\Sigma^1| = |\Sigma^2| = k$ *and there exist bijections:* $\Pi_1 : \Sigma^1 \rightarrow [0, k)$, $\Pi_2 : \Sigma^2 \rightarrow [0, k)$ *and bijection* $\beta : Q^1 \rightarrow Q^2$ *such that* $\beta(q_0^1) = q_0^2$ *and, for all* $\sigma^1 \in \Sigma^1$ *and* $q^1 \in Q^1$, $\beta(\delta(q^1, \sigma^1) = \delta^2(\beta(q^1), \Pi_2^{-1}(\Pi_1(\sigma^1))))$, *and* $\beta(F^1) = F^2$ *[39]*

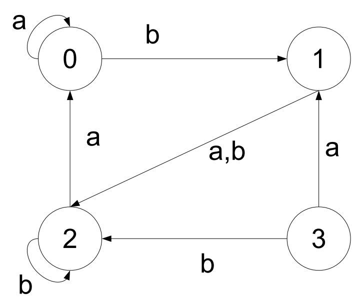


Figure 2.4 State '3' is unreachable.

2.1.3. Finite state transducer

Generalizing the knowledge about FA (see Section 2.1.2), we can construct a machine, which produces not only one output value (accepted or rejected), but the sequence of output values, i.e. an output string (see Figure 2.5). While there are several types of FST, we consider only two of them – the Moore machine and the Mealy machine.

Definition 2.25 (Moore machine) *is a six-tuple* $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$*, wherein*

- Q *is a finite set of states and* q_0 *denotes the start state,*
- Σ *is the input alphabet,*
- Δ *is the output alphabet,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function,*
- $\lambda : Q \times \Sigma \to \Delta$ *is the output function represented by the output table that shows what character from* Δ *is printed by each state that is entered [38].*

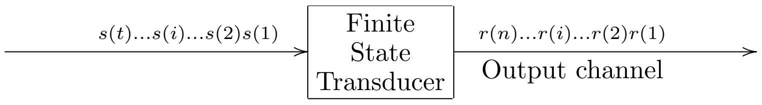


Figure 2.5 Conception of FST

The work-flow for the Moore machine is presented in Algorithm 2.

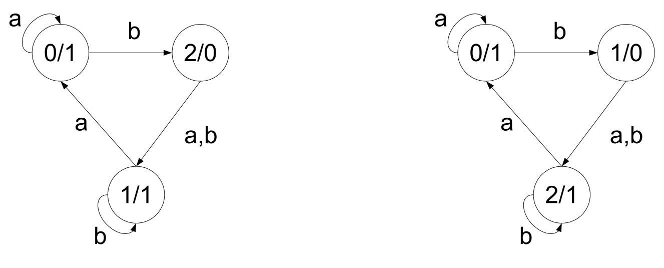
Algorithm 2 Moore machine work-flow

```
1: function runMachine($inputString$)
2:     $outputString \leftarrow empty$
3:     $q_c \leftarrow q_0$
4:     $outputString \leftarrow outputString + q_c.value$
5:     for $i = 0 \rightarrow$ length($inputString$) $- 1$ do
6:         makeTransition($inputString[i]$)
7:     end for
8: end function

9: function makeTransition($ic$)
10:     $transition \leftarrow TransitionTable[q_c.label][\Sigma.position(ic)]$
11:     $q_c \leftarrow q_{transition.toState}$
12:     $outputString \leftarrow outputString + q_{transition.toState}.value$
13: end function
```

Example 2.5 *Figure 2.6 illustrates two Moore machines. We can construct the function for translating the states of Moore machine M_{iso1} to the states of Moore machine M_{iso2}:*

$$M_{iso1}.q_0 \rightarrow M_{iso2}.q_0$$
$$M_{iso1}.q_1 \rightarrow M_{iso2}.q_2$$
$$M_{iso1}.q_2 \rightarrow M_{iso2}.q_1$$

Thus, Moore machine M_{iso1} is isomorphic to M_{iso2}.



(a) Moore machine M_{iso1} (b) Moore machine M_{iso2}

Figure 2.6 Two isomorphic Moore machines

Definition 2.26 (Mealy machine) *is a six-tuple* $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$*, wherein:*

- *Q is a set of states, where q_0 denotes the start state,*
- *Σ is the input alphabet,*
- *Δ is the output alphabet,*
- *$\delta : Q \times \Sigma \to Q$ is the transition function, and*
- *$\lambda : Q \times \Sigma \to \Delta$ the output function represented by the output table that shows what character from Δ is printed by each transition that is processed [38].*

The work-flow for the Mealy machine is presented in Algorithm 3.

Algorithm 3 Mealy machine work-flow

```
 1: function runMachine($inputString$)
 2:     $outputString \leftarrow empty$
 3:     $q_c \leftarrow q_0$
 4:     for $i = 0 \to$ length(inputString) $- 1$ do
 5:         makeTransition($inputString[i]$)
 6:     end for
 7: end function

 8: function makeTransition($ic$)
 9:     $transition \leftarrow TransitionTable[q_c.label][\Sigma.position(ic)]$
10:     $q_c \leftarrow q_{transition.toState}$
11:     $outputString \leftarrow outputString + transition.value$
12: end function
```

Definition 2.27 (Input/output sequence (I/O sequence)) *An input-output sequence S of length n is a set of pairs $\{(i_0, o_0), (i_1, o_1), \ldots, (i_n, o_n)\}$ where $(i_i, o_i) \in \Sigma \times \Delta$. An input/output sequence set ζ is a set of S.*

The Moore and Mealy machines behave differently while processing output: Moore machines print a character when entering the state, while Mealy machines print a character when traversing an arc. However it is possible to construct equivalent machines.

Theorem 2.2 *If M_o is a Moore machine, then there is a Mealy machine M_e that is equivalent to it. For every Mealy machine M_e, there is an equivalent Moore machine M_o.*

Definition 2.28 (Equivalence of Mealy and Moore machines) *Given a Mealy machine M_e and a Moore machine M_o, which automatically prints the character $'x'$ in the start state, we say that these two machines are equivalent, if for every input string the output string from M_o is exactly $'x'$ concatenated with output from M_e.*

2.1.4. Package: Finite state machine

This package contains structures and algorithms required for correctly working with FSMs. The main class of the package is `FSM`, which represents the structure, while other classes define the parts of FSM.

- `State` class implements the attributed states (see Definition 2.35).
- `Transition` class implements the attributed transition.
- `Alphabet` class implements the alphabet structure (see Definition 2.1).
- `FSM` class presents the abstract structure for storing FSM (see Definitions 2.26, 2.25). The `FSM` class is consists of a set of states, a set of transitions and a transition table, which defines the relation between states and transition. `FSM` class also contains the input and output alphabets.

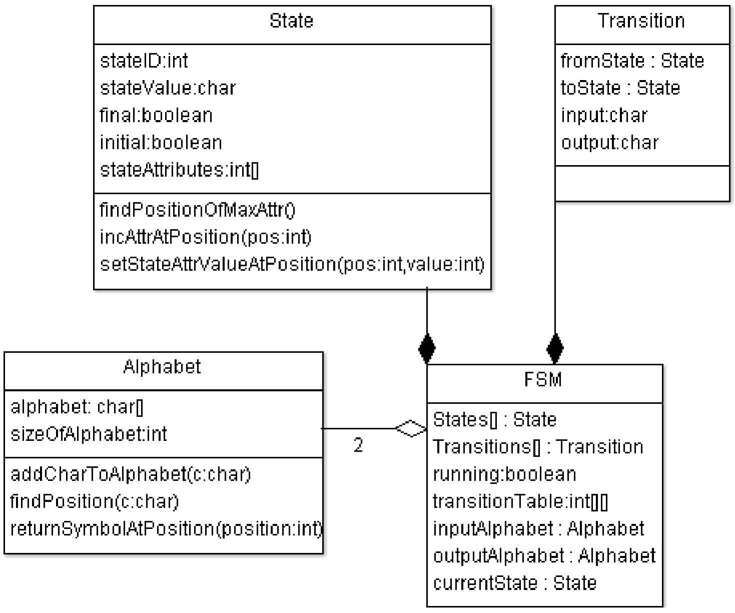


Figure 2.7 Class diagram: FSM

2.2. String Representation

This section discusses the different representations of a FST. The proper representation of a FST is essential to the optimization of a search algorithm. Each representation of a FST must allow for:

- correct transformations $[FST \rightarrow \mathit{Representation}(FST)]$ and vice versa $[\mathit{Representation}(FST) \rightarrow FST]$,
- random generation of points in search space and
- operations on the points that store their properties and correctness.

2.2.1. Existing solutions

There are several FSM representations, which are used for different heuristic search algorithms. Mainly, they can be divided into two groups:

I **FSM is represented by a transition diagram** (see Definition 2.16). For example in [6], [20], [32], [26], [40] The general idea of this representation method lies in the elimination of transformations: $[FSM \rightarrow \mathit{Representation}(FSM)]$ and $[\mathit{Representation}(FSM) \rightarrow FSM]$. The search space is defined so, that solutions are transition diagrams (graphs) and all movements in the search space are defined on the graph modification level. This representation is used together with Genetic Programming.

II **FSM is represented by a table** (see Definition 2.18). This representation method is based on tabular FSM representation. Thus, each point of the search space is a structure, which can be considered as a transition and output table. There are three main subgroups:

- **2D representation.** In [5], [33], [31], [34], [18] a FSM is represented by a transition and output table. The transition and output tables are used directly, therefore the solution transformations during the search algorithm work on the table modification level.
- **1D representation.** In [27], [15] a FSM is represented in tabular form, but transition and output table is transformed into a linear 1D structure, for example by the concatenation of rows.
- **Bit string representation.** In [19], [13] a FSM is represented by binary string.This way, the transition and output table is coded by binary numbers.

2.2.2. Restrictions of the search space

We are interested only in tabular representation of FSM (in 1D form). In the following subsection, the basic definitions and algorithms required for FSM transformation are introduced.

To minimize the size of the search space, we consider only FSTs with the following properties:

- **deterministic**, i.e. for each state and input symbol there is only transition from this state labeled by this symbol. This also requires there to only be one initial state,
- **completeness**,i.e. for each state and symbol in the input alphabet there is a transition from this state labeled by this symbol,
- **initial state is labeled** by '0',
- **known number of states** $|Q| = n$,
- $\epsilon-$ **transitions** are not allowed
- $\epsilon-$ **output values** are not allowed.

2.2.3. Proposed solutions

We propose to only use string representations of FST. String representation allows to apply different stochastic optimization methods, which work in the discrete search space, to the problem of FST inference. Nevertheless, for correctly working with the corresponding string representation of FSM, both transformation algorithms are required $[FST \to Representation(FST)]$ and $[Representation(FST) \to FST]$, which requires additional computational power.

Figure 2.8 shows the relation between different string representations introduced in the following sections:

- Direct concatenation of a transition table (see Section 2.3): $\mathrm{SR(FST)}$ – string representation of FST (see Subsection 2.3.1), $\mathrm{SR^B(FST)}$ – binary string representation of FST (see Subsection 2.3.2). These string representations were initially proposed in the master thesis of the author [41]. $\mathrm{SR^B(FST)}$ is also presented in [42].
- Separating the transition and output functions of FST. Deriving the output function (see Section 2.4): $\mathrm{SR_S(FST)}$ – string representation of FST with a separated output function (see Subsection 2.4.1), $\mathrm{SR_D(FST)}$ – string representation of FST with a derived output function (see Subsection 2.4.2). $\mathrm{SR_S(FST)}$ is a further development of $\mathrm{SR(FST)}$.

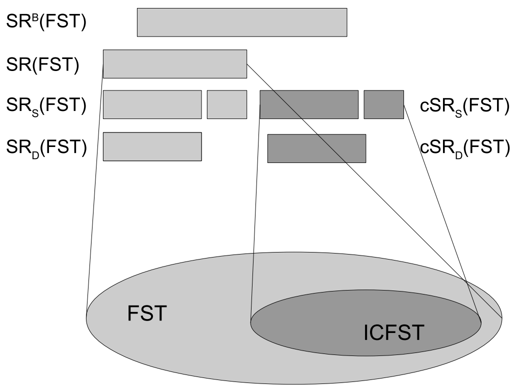


Figure 2.8 Different FST representation codes

- Canonical string representation: $\mathrm{cSR_S(FST)}$ – canonical string representation of FST (see Subsection 2.7.2), $\mathrm{cSR_D(FST)}$ – canonical string representation of FST with a derived output function (see Subsection 2.7.3). $\mathrm{cSR_S(FST)}$ is an improvement of $\mathrm{SR_S(FST)}$, which applies the normal form string representation. This approach allows to eliminate isomorphisms and FSMs with unreachable states, which makes the search space smaller.

2.3. Direct Concatenation of a Transition Table

The general idea behind this type of representation is the transformation of a transition table into string code, which stores all he information about transitions and output functions.

2.3.1. $\mathrm{SR(FST)}$: String representation of FST

The intuitive way of transforming $[Transition\ table \to string]$ is to present the transition table as the concatenation of its rows. This subsection covers the definitions as well as the encoding and decoding transformation algorithms for Moore and Mealy machines.

Moore machine

Taking a target Moore machine with n states, where:

- the input alphabet is $\Sigma = \{i_0, \ldots, i_{k-1}\}$,
- the output alphabet is $\Delta = \{o_0, \ldots, o_{m-1}\}$,
- the set of states is $Q = \{q_0, \ldots, q_{n-1}\}$.

The row of the transition table, which contains information about one state q_j can be described by the structure presented in Figure 2.9. This structure contains o^j as a code for the corresponding output symbol at the state and $q^{i_{\cdots}}$ are codes (labels) for target transition states with respect to the input symbol $i_{0\ldots k-1}$.

State q_j			
o^j	q^{i_0}	$q^{\cdots}$	$q^{i_{k-1}}$

Figure 2.9 One row of a Moore machine transition table

In order to transform the entire transition table into the string, we concatenate the rows of the transition table in the order of state labels. Figure 2.10 shows the resulting structure of the Moore machine string representation.

State q_0				$\ldots$				State q_{n-1}			
o^0	$q_0^{i_0}$	$q_0^{i_{\ldots}}$	$q_0^{i_{k-1}}$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	o^{n-1}	$q_{n-1}^{i_0}$	$q_{n-1}^{i_{\ldots}}$	$q_{n-1}^{i_{k-1}}$

Figure 2.10 SR(MoFST) *as a concatenation of rows of a Moore machine transition table*

Definition 2.29 (String representation of a Moore machine) *is a structure in the form*

$$SR(MoFST) = [o^0 q_0^{i_0} q_0^{i_{\cdots}} q_0^{i_{k-1}} \ldots o^{n-1} q_{n-1}^{i_0} q_{n-1}^{i_{\ldots}} q_{n-1}^{i_{k-1}}],$$

where

- $[o^0 \ldots o^{n-1}] \in [0 \ldots m-1]$ *present codes for output values and*
- $[q^{i_0} \ldots q_{n-1}^{i_{k-1}}] \in [0 \ldots n-1]$ *present target states of the transitions.*

Theorem 2.3 (The space complexity of SR(MoFST)) *The length of* SR(MoFST) *for a Moore machine with n states and over the input alphabet k with the output alphabet with m symbols is*

$$(1+k) \times n$$

The number of corresponding SR(MoFST) *strings is*

$$(m \times n^k)^n$$

Example 2.6 *A Moore machine M_{mo1} with a transition diagram represented in Figure 2.11 is examined more closely.*

For example, let the machine M_{mo1} have 4 states $Q = \{0,1,2,3\}$, the input alphabet contain 2 symbols $\Sigma = \{a,b\}$ and the output alphabet 2 symbols $\Delta = \{0,1\}$. Then the Moore machine M_{mo1} can be represented as $SR(M_{mo1}) = [1,1,0,1,1,2,0,3,3,1,1,0]$ (see Figure 2.12).

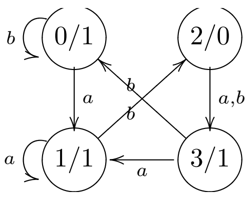


Figure 2.11 Transition diagram for a Moore machine M_{mo1}

	a	b		a	b		a	b		a	b
1	1	0	1	1	2	0	3	3	1	1	0

Figure 2.12 String representation $SR(M_{mo1})$

The transformation $[Moore\ machine \to SR(MoFST)]$ is given by Definition 2.29. The transformation $[SR(MoFST) \to Moore\ machine]$ is presented by Algorithm 4.

Algorithm 4 $SR(MoFST) \to Moore\ machine$ transformation

```
Require: Input alphabet $\Sigma$ with $k$ symbols
Require: Input alphabet $\Delta$ with $m$ symbols
Require: SR(MoFST)
 1: for $i = 0 \to n - 1$ do
 2:     createState($q_i$)
 3:     $q_i.value \leftarrow \Delta_{position(SR[i \times (1+k)])}$
 4:     if $i == 0$ then
 5:         setInitial($q_i$)
 6:     end if
 7: end for
 8: for $i = 0 \to n - 1$ do
 9:     $fromState \leftarrow q_i$
10:     for $j = 0 \to k - 1$ do
11:         $toState \leftarrow q_{SR[i \times (1+k)+j+1]}$
12:         $inSymbol \leftarrow \Sigma_j$
13:         $transition \leftarrow Transition(fromState, toState, inSymbol)$
14:     end for
15: end for
16: return Moore machine
```

The generator of a random Moore machine in the form of SR(MoFST) is presented in Algorithm 5.

Algorithm 5 Random generator of SR(MoFST)

```
Require: Input alphabet $\Sigma$ with $k$ symbols
Require: Input alphabet $\Delta$ with $m$ symbols
1: for $i = 0 \to n - 1$ do
2:     $SR[i \times (1 + k)] \leftarrow$ randomInteger($0 \ldots m - 1$)
3:     for $j = 0 \to k - 1$ do
4:         $SR[i \times (1 + k) + j + 1] \leftarrow$ randomInteger($0 \ldots n - 1$)
5:     end for
6: end for
7: return $SR(MoFST)$
```

Mealy machine

Taking a target Mealy machine with n states, where:

- the input alphabet is $\Sigma = \{i_0, \ldots, i_{k-1}\}$,
- the output alphabet is $\Delta = \{o_0, \ldots, o_{m-1}\}$,
- the set of states is $Q = \{q_0, \ldots, q_{n-1}\}$.

One row of the transition table for such a Mealy machine can be described by the structure presented in Figure 2.13. There the information for the corresponding state q_j is stored. For all transitions from this state with respect to the input symbol $i_{0 \ldots k-1}$ we encode the output value $o^{i_{\cdots}}$ and the label of target state $q^{i_{\cdots}}$.

State q_j					
o^{i_0}	q^{i_0}	$o^{i_{\cdots}}$	$q^{i_{\cdots}}$	$o^{i_{k-1}}$	$q^{i_{k-1}}$

Figure 2.13 One row of a Mealy machine transition table

The structure, required for coding the entire transition table is constructed as the concatenation of sections (see Figure 2.13) in the order of state labels (see Figure 2.14).

State q_0					$\ldots$	State q_{n-1}				
$o_0^{i_0}$	$q_0^{i_0}$	$\ldots$	$o_0^{i_{k-1}}$	$q_0^{i_{k-1}}$	$\ldots$	$o_{n-1}^{i_0}$	$q_{n-1}^{i_0}$	$\ldots$	$o_{n-1}^{i_{k-1}}$	$q_{n-1}^{i_{k-1}}$

Figure 2.14 SR(MeFST) *as the concatenation of the Mealy machine transition table rows*

Definition 2.30 (String representation of a $MeFST$) *is the structure in the following form:*

$$SR(MeFST) = [o_0^{i_0} q_0^{i_0} \dots o_0^{i_{k-1}} q_0^{i_{k-1}} \dots o_{n-1}^{i_0} q_{n-1}^{i_0} \dots o_{n-1}^{i_{k-1}} q_{n-1}^{i_{k-1}}],$$

where

- $[o_0^{i_0} \dots o_{n-1}^{i_{k-1}}] \in [0 \dots m-1]$ *present codes for the output values of the transitions and*
- $[q_0^{i_0} \dots q_{n-1}^{i_{k-1}}] \in [0 \dots n-1]$ *present the target states of the transitions.*

Theorem 2.4 (The space complexity of $\mathrm{SR(MeFST)}$) *The length of* $\mathrm{SR(MeFST)}$*, which represents a Mealy machine with* n *states and over input alphabet* k *and with output alphabet with* m *symbols is*

$$((1+1) \times k) \times n$$

The number of corresponding $\mathrm{SR(MeFST)}$ *strings is*

$$((m \times n)^k)^n.$$

The transformation $[Mealy\ machine \rightarrow SR(MeFST)]$ is given by Definition 2.30. The transformation $[SR(MeFST) \rightarrow Mealy\ machine]$ is presented by Algorithm 6.

Example 2.7 *A Mealy machine* M_{me1} *with the transition diagram represented in Figure 2.15 is considered further.* $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$. *The string representation for this Mealy machine is*
$SR(M_{me1}) = [1, 2, 0, 0, 0, 0, 1, 1, 1, 3, 0, 3, 1, 0, 0, 1]$ *(see Figure 2.16).*

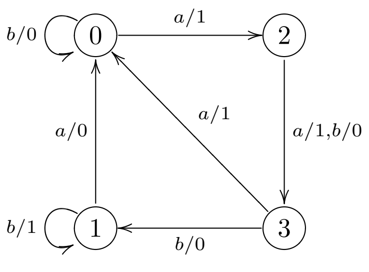


Figure 2.15 Transition diagram of Mealy machine M_{me1}

State 0				State 1				State 2				State 3			
a		b		a		b		a		b		a		b	
1	2	0	0	0	0	1	1	1	3	0	3	1	0	0	1

Figure 2.16 String representation $SR(M_{ex2})$

Algorithm 6 $SR(MeFST) \rightarrow Mealy\ machine$ transformation

```
Require: Input alphabet $\Sigma$ with $k$ symbols
Require: Input alphabet $\Delta$ with $m$ symbols
Require: SR(MeFST)
 1: for $i = 0 \rightarrow n - 1$ do
 2:     createState($q_i$)
 3:     if $i == 0$ then
 4:     end if
 5: end for
 6: for $i = 0 \rightarrow n - 1$ do
 7:     $fromState \leftarrow q_i$
 8:     for $j = 0 \rightarrow k - 1$ do
 9:         $toState \leftarrow q_{SR[i \times 2 \times k + j \times 2 + 1]}$
10:         $inSymbol \leftarrow \Sigma_j$
11:         $outSymbol \leftarrow \Delta_{SR[i \times 2 \times k + j \times 2]}$
12:         $transition \leftarrow Transition(fromState,$
    $toState, inSymbol, outSymbol)$
13:     end for
14: end for
15: return Mealy machine
```

2.3.2. $\mathrm{SR}^{\mathrm{B}}(\mathrm{FST})$: Binary string representation of FST

Some of heuristic search algorithms, such as the Genetic Algorithm, only work with binary multidimensional search spaces. Therefore, if we apply such heuristic search algorithms to the problem of FSM identification, we need to construct a method for representing a FSM in the form of binary strings.

The intuitive way of representing FSM as binary string is to apply encoding methods presented in Subsection 2.3.1: $[FST \rightarrow SR(FST)]$ and transform each integer of the code $SR(FST)$ into binary string $[SR(FST) \rightarrow SR^B(FST)]$ (see Algorithm 7).

Algorithm 7 $SR(FST) \rightarrow SR^B(FST)$ transformation

```
Require: SR(FST)
  for all $integer\ i \in SR(FST)$ do
      $b \leftarrow$ integerToBinary($i$)
      $SR^B(FST) \leftarrow SR^B(FST)$+$b$
  end for
  return $SR^B(FST)$
```

Definition 2.31 ($\mathrm{SR^B(FST)}$: Binary string representation) *Binary string representation of* FST *is a string constructed by consequent transformation of each integer in* $\mathrm{SR(FST)}$ *to binary string.*

Example 2.8 *Let us consider a Moore machine M_{mo1} (see Example 2.6) and a Mealy machine M_{me1} (see Example 2.7) and transform their* $\mathrm{SR(FST)}$ *codes into binary strings. The corresponding binary representation of M_{mo1} is $SR^B(M_{mo1})$ =* [10100101100111110100] *(see Figure 2.17) and the corresponding binary representation of M_{me1} is $SR^B(M_{me1})$ =* [110000000101111011100001] *(see Figure 2.18)*

	a	b		a	b		a	b		a	b	
1	1	0	1	1	2	0	3	3	1	1	0	$SR()$
1	01	00	1	01	10	0	11	11	1	01	00	$SR^B()$

Figure 2.17 Binary string representation $SR^B(M_{mo1})$

State 0				State 1				State 2				State 3			
a		b		a		b		a		b		a		b	
1	2	0	0	0	0	1	1	1	3	0	3	1	0	0	1
1	10	0	00	0	00	1	01	1	11	0	11	1	00	0	01

Figure 2.18 Binary string representation $SR^B(M_{me1})$

Theorem 2.5 *For the Moore machine with n states, where:*

- *the input alphabet is $\Sigma = \{i_0, \ldots, i_{k-1}\}$,*
- *the output alphabet is $\Delta = \{o_0, \ldots, o_{m-1}\}$,*
- *the set of states is $Q = \{q_0, \ldots, q_{n-1}\}$.*

In order to present one symbol of Δ we require $\lceil log_2 m \rceil$ bits, while for encoding one state q the $\lceil log_2 n \rceil$ bits are required. Thus, the length of $\mathrm{SR^B(MoFST)}$ *can be computed by:*

$$|SR^B(MoFST)| = n \times (\lceil log_2 m \rceil + k \times \lceil log_2 n \rceil).$$

The transformation $[SR(MoFST) \rightarrow SR^B(MoFST)]$ is described by Definition 2.31, the reverse transformation $[SR^B(MoFST) \rightarrow SR(MoFST)]$ is described by Algorithm 8.

Algorithm 8 $SR^B(MoFST) \to SR(MoFST)$ transformation

```
Require: $SR^B(MoFST)$
 1: $bO \leftarrow \lceil log_2 m \rceil$
 2: $bN \leftarrow \lceil log_2 n \rceil$
 3: for $i = 0 \to n-1$ do
 4:     $booleanString \leftarrow empty$
 5:     for $j = 0 \to bO-1$ do
 6:         $booleanString \leftarrow booleanString + SR^B(MoFST)[i \times (bO + k \times bN) + j]$
 7:     end for
 8:     $SR(MoFST)[i \times (1+k)] \leftarrow$ parse($booleanString$)
 9:     $booleanString \leftarrow empty$
10:     for $j = 0 \to k-1$ do
11:         for $k = 0 \to bN-1$ do
12:             $booleanString \leftarrow booleanString + SR^B(MoFST)[i \times (bO + k \times bN) + j \times bN + k + bO]$
13:         end for
14:         $SR(MoFST)[i \times (1+k) + j + 1] \leftarrow$ parse($booleanString$)
15:         $booleanString \leftarrow empty$
16:     end for
17: end for
18: return SR(MoFST)
```

Theorem 2.6 *Similar computations can be done for the Mealy machine with n states, where*

- *the input alphabet is $\Sigma = \{i_0, \dots, i_{k-1}\}$,*
- *the output alphabet is $\Delta = \{o_0, \dots, o_{m-1}\}$,*
- *the set of states is $Q = \{q_0, \dots, q_{n-1}\}$.*

In order to present one symbol of Δ we require $\lceil log_2 m \rceil$ bits, while for encoding one state q the $\lceil log_2 n \rceil$ bits are required. Thus, the length of $\mathrm{SR^B(MeFST)}$ *can be computed by:*

$$|SR^B(MeFST)| = n \times k \times (\lceil log_2 m \rceil + \lceil log_2 n \rceil)$$

The transformation $[SR(MeFST) \to SR^B(MeFST)]$ is described by Definition 2.31, the reverse transformation $[SR^B(MeFST) \to SR(MeFST)]$ is described by Algorithm 9.

Although the encoding process $[SR(FST) \to SR^B(FST)]$ is correct, the reverse process $[SR^B(FST) \to SR(FST)]$ can be incorrect in situations where $SR^B(FST)$ was generated. For example, for a random binary string with

Algorithm 9 $SR^B(MeFST) \to SR(MeFST)$ transformation

```
Require: $SR^B(MeFST)$
 1: $bO \leftarrow \lceil log_2 m \rceil$
 2: $bN \leftarrow \lceil log_2 n \rceil$
 3: for $i = 0 \to n \times k - 1$ do
 4:     $booleanString \leftarrow empty$
 5:     for $j = 0 \to bO - 1$ do
 6:         $booleanString \leftarrow booleanString + SR^B(MeFST)[i \times (bO + bN) + j]$
 7:     end for
 8:     $SR(MeFST)[2 \times i] \leftarrow$ parse($booleanString$)
 9:     $booleanString \leftarrow empty$
10:     for $j = 0 \to bN - 1$ do
11:         $booleanString \leftarrow booleanString + SR^B(MeFST)[i \times (bO + bN) + bO + j]$
12:     end for
13:     $SR(MeFST)[2 \times i + 1] \leftarrow$ parse($booleanString$)
14: end forreturn SR(MeFST)
```

dimensionality $|SR^B(FST)|$, not all binary codes have corresponding integers inside the range (see Example 2.9).

Example 2.9 *Let a Moore machine M_{in} have 3 states $Q = \{0, 1, 2\}$. Two bits are required for storing information about states. During the decoding process a pair of bits is interpreted as shown in Figure 2.19. The Moore machine represented by such* $\mathrm{SR^B(FST)}$ *is invalid.*

bits	state
00	q_0
01	q_1
10	q_2
11	error

Figure 2.19 $\mathrm{SR^B(FST)}$. *Error in the decoding process*

Such invalid $\mathrm{SR^B(FST)}$ representations (see Example 2.9) can appear during a random generation of FST in $\mathrm{SR^B(FST)}$ form or after some transformations of $\mathrm{SR^B(FST)}$ due to certain search algorithm operations.

In order to solve the problem of invalid $\mathrm{SR^B(FST)}$ we need to modify $[SR^B(FST) \to FST]$ transformation. For example one of the following methods can be applied:

- **Ignore** method, where an invalid FSM is marked as an incorrect machine and no longer considered in a search process.
- **Repair** method, where a non-existing codes are translated into existing labels, or replaced by existing codes.
- **Generate a new** $\mathrm{SR^B(FST)}$ method, which can be applied only during the random generation of the machines. If a generated FSM is invalid, then a new $\mathrm{SR^B(FST)}$ must be generated, until the generated FSM is correct.
- **Restrict** method, where the number of states n and the sizes of input and output alphabets are restricted by the powers of two (see [43]).

Further information about $\mathrm{SR^B(FST)}$ can be found in [41], where $\mathrm{SR^B(FST)}$ was used with combination with a GA and in Publication C [42], where a binary GSA was used as a search algorithm.

2.4. Separating a FST Structure. Deriving the Output Function from the Training Set

The $\mathrm{SR(FST)}$ coding scheme is very powerful as it contains both transition and output functions. This allows to control all of the FST properties. Nevertheless, it should be observed that the search space contains FSTs with the same transition function, but with different output functions (see Example 2.10). In the context of search algorithm this means that it is more difficult to find both the correct transition and the output function. However, if we separate the output and transition functions into two different structures and apply a search algorithm only to the transition function part (the output function can be derived from the training set), we can minimize the space complexity.

Example 2.10 *Let us have a Moore machine M_{aab} with 4 states $\Sigma = \{b, a\}$ and $\Delta = \{0, 1\}$. Our task is to construct Moore machine, which is consistent with the training set 'aab recognizer' (Figure 4.8).*

Figure 2.20 illustrates certain chosen points of the search space, where each point is $SR(M_{aab})$ code. Each point clearly presents M_{aab} with the same transition function, but with the all possible combinations of output functions. M_{aab} contains four states, and each state can have output value from Δ, resulting in 4^2 possible combinations. Each point was evaluated with respect to the 'aab recognizer' training set. Evidently, only one point with $SR(M_{aab}) = [0, 0, 1, 0, 0, 2, 0, 3, 2, 1, 0, 1]$ is a valuable in the context of this training set. Other points can be omitted.

Definitions of the coding scheme $\mathrm{SR_S(FST)}$, where transition and output functions are considered separately are presented in the Subsection 2.4.1. Subsection 2.4.2 defines the concepts of *attributed state* and *attributed transition* and provides algorithms for constructing Moore and Mealy machines through *decorating* $\mathrm{DFA}_{\emptyset}$, using the training set.

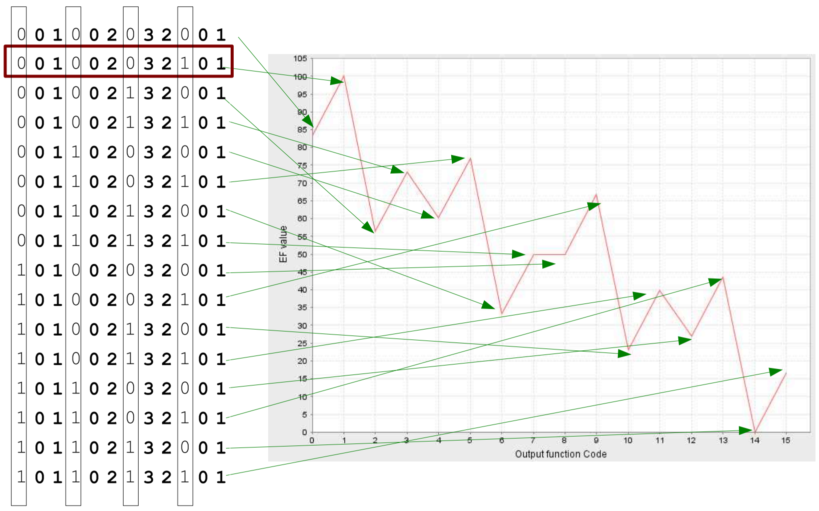


Figure 2.20 All possible combinations for the output function with a predefined structure

2.4.1. $\mathrm{SR_S(FST)}$: Separating the structure of the FST from the output function

$\mathrm{SR_S(MoFST)}$: a Moore machine

The separation of the transition and output functions can easily be done by constructing two strings from a $\mathrm{SR(FST)}$ string. Figure 2.21 depicts the separation of the transition and output function transformation for MoFST.

Definition 2.32 (Separated string representation of Moore machine) *is a structure derived from the* $\mathrm{SR(MoFST)}$*:*

$$SR_S(MoFST) = \{SR_S(MoFST.transition), SR_S(MoFST.output)\},$$

where

- $SR_S(MoFST.transition)$ *is a transition function* $q_0^{i_0} q_0^{i_{\dots}} q_0^{i_{k-1}} \dots q_{n-1}^{i_0}$ $q_{n-1}^{i_{\dots}} q_{n-1}^{i_{k-1}}$*, where* $[q^{i_0} \dots q_{n-1}^{i_{k-1}}] \in [0 \dots n-1]$ *presents the target states of the transitions and*
- $SR(MoFST.output)$ *is an output function* $o^0 \dots o^{n-1}$*, where* $[o^0 \dots o^{n-1}] \in [0 \dots m-1]$ *presents codes for the output values of the Moore machine.*

State q_0				$\dots$			State q_{n-1}			
o^0	$q_0^{i_0}$	$q_0^{i_{\dots}}$	$q_0^{i_{k-1}}$	$\dots$	$\dots$	$\dots$	o^{n-1}	$q_{n-1}^{i_0}$	$q_{n-1}^{i_{\dots}}$	$q_{n-1}^{i_{k-1}}$

(a) $SR_S(MoFST)$

State q_0			$\dots$			State q_{n-1}		
$q_0^{i_0}$	$\dots$	$q_0^{i_{k-1}}$	$\dots$	$\dots$	$\dots$	$q_{n-1}^{i_0}$	$\dots$	$q_{n-1}^{i_{k-1}}$

(b) $SR_S(MoFST.transition)$

State q_0	$\dots$	State q_{n-1}
o^0	$\dots$	o^{n-1}

(c) $SR_S(MoFST.output)$

Figure 2.21 $\mathrm{SR_S(MoFST)}$*: Separating transition and output functions for a Moore machine*

There is a transformation $SR(MoFST) \to SR_S(MoFST)$ and vice versa, therefore these coding systems are isomorphic (Theorem 2.7).

Theorem 2.7 *The* $\mathrm{SR_S(MoFST)}$ *is isomorphic to* $\mathrm{SR(MoFST)}$.

Proof. *There are direct transformations $SR(MoFST) \to SR_S(MoFST)$ and $SR_S(MoFST) \to SR(MoFST)$*

The structure defined by $SR_S(MoFST.transition)$ can be directly transformed into $\mathrm{DFA}_\emptyset$. Thus the transition table of a Moore machine can be processed as $DFA_\emptyset$ (Theorem 2.8).

Theorem 2.8 $SR_S(MoFST.transition) \to DFA_\emptyset$.

Example 2.11 *Let us consider M_{mo1} (see Example 2.6). The original code is $SR(M_{mo1}) = [1,1,0,1,1,2,0,3,3,1,1,0]$. Hence, the result of the separated string representation is: $SR_S(M_{mo1}) = \{[1,0,1,2,3,3,1,0],[1,1,0,1]\}$.*

$\mathrm{SR_S(MeFST)}$**: a Mealy machine**

Similar transformations can be done for a Mealy machine (see Figure 2.22).

Definition 2.33 (Separated string representation of Mealy machine) *is a structure formed from* $\mathrm{SR(MeFST)}$*:*

$$SR_S(MeFST) = \{SR_S(MeFST.transition), SR_S(MeFST.output)\},$$

where:

- $SR_S(MeFST.transition)$ *is a transition function* $q_0^{i_0} \dots q_0^{i_{k-1}} \dots q_{n-1}^{i_0} \dots q_{n-1}^{i_{k-1}}$, *where* $[q^{i_0} \dots q_{n-1}^{i_{k-1}}] \in [0 \dots n-1]$ *presents target states of the transitions and*

State q_0					$\dots$	State q_{n-1}				
$o_0^{i_0}$	$q_0^{i_0}$	$\dots$	$o_0^{i_{k-1}}$	$q_0^{i_{k-1}}$	$\dots$	$o_{n-1}^{i_0}$	$q_{n-1}^{i_0}$	$\dots$	$o_{n-1}^{i_{k-1}}$	$q_{n-1}^{i_{k-1}}$

(a) $SR(MeFST)$

State q_0			$\dots$	State q_{n-1}		
$q_0^{i_0}$	$\dots$	$q_0^{i_{k-1}}$	$\dots$	$q_{n-1}^{i_0}$	$\dots$	$q_{n-1}^{i_{k-1}}$

(b) $SR_S(MeFST.transition)$

State q_0			$\dots$	State q_{n-1}		
$o_0^{i_0}$	$\dots$	$o_0^{i_{k-1}}$	$\dots$	$o_{n-1}^{i_0}$	$\dots$	$o_{n-1}^{i_{k-1}}$

(c) $SR_S(MeFST.output)$

Figure 2.22 $\mathrm{SR_S(MeFST)}$*: Separating transition and output functions for a Mealy machine*

- *$SR_S(MeFST.output)$ is an output function $o_0^{i_0} \dots o_0^{i_{k-1}} \dots o_{n-1}^{i_0} \dots o_{n-1}^{i_{k-1}}$, where $[o^0 \dots o^{n-1}] \in [0 \dots m-1]$ presents codes for output values of Mealy machine.*

Theorem 2.9 *The* $\mathrm{SR_S(MeFST)}$ *is isomorphic to* $\mathrm{SR(MeFST)}$.

Proof. *There are direct transformations $SR(MeFST) \rightarrow SR_S(MeFST)$ and $SR_S(MeFST) \rightarrow SR(MeFST)$*

The structure defined by $SR_S(MeFST.transition)$ can be directly transformed into $\mathrm{DFA}_\emptyset$. Thus a transition table of a MeFST can be processed as $\mathrm{DFA}_\emptyset$ (Theorem 2.10).

Theorem 2.10 $SR_S(MeFST.transition) \rightarrow DFA_\emptyset$

Example 2.12 *Let us consider M_{me1} (see Example 2.7). The original code is $SR(M_{me1}) = [1,2,0,0,0,0,1,1,1,3,0,3,1,0,0,1]$. Hence, the separated string representation is: $SR_S(M_{me1}) = \{[2,0,0,1,3,0,3,0,0,1],[1,0,0,1,1,0,1,0]\}$*

2.4.2. $\mathrm{SR_D(FST)}$: Deriving the output function

For a given FST structure (transition function) and training data, which consists of I/O pairs, we can construct a FST output function that processes the training set in the most correct way. Naturally, this method can only be applied in situations where the output strings are directly observable.

Existing solutions

There are several methods for deriving output functions for different types of FSMs based on the statistical analysis of training data. Some examples are:

- **Deriving the output function for a Moore machine** ([19]). The method is also based on constructing the counting table, where for each state there is a row with numbers, which corresponds to a certain symbol from the output alphabet. Thus, the algorithm increases a specific number in this table, if the Moore machine is required to output a corresponding symbol (observed from the output of the training set) for a given input string. The output value at the state is the symbol corresponding to the maximum number in the row.

- **Deriving the output function for a Mealy machine** ([17], [43]). The idea is very similar, but in this case the table contains a number for each transition/output symbol pair. The update process of the table is also based on observations from the output string and increasing the corresponding number.

- **Deriving 'final/not final' labels for** FA ([7], [44]). The algorithms *Smart State Labeling* and *Smart Tuning the Output Labels* construct the table, where there are two numbers for each state. If the finite acceptor in this state stopped working for certain input from the training set and this string is marked as positive, then we increase the number in table which corresponds to 'accepted'. However, if the string is marked as negative, we increase the number, which denotes the 'negative' label. The label 'final/not final' state of the finite acceptor is defined by the maximum number in the corresponding row of the count table for this state.

We propose the generalization of the idea of deriving the output function from the training set through integrating 'counting' tables into the FST structure. This allows to construct effective algorithms which are similar to the original FST work-flow.

Deriving the output function of a Moore machine

Subsection presents the methods and definitions for deriving the output function from the training set in the Moore machine example.

Definition 2.34 (A Moore machine with the derived output function)

$$SR_D(MoFST) = \{SR_D(MoFST.transition), MoFST.output\}$$

$\mathrm{SR_D}(\mathrm{MoFST})$ *string representation is derived from* $\mathrm{SR_S}(\mathrm{MoFST})$*, where*

- $SR_D(MoFST.transition) \leftarrow SR_S(MoFST.transition)$ *and*
- $MoFST.output$ *is constructed by Algorithm 10.*

The behavior of the output function for the Moore machine depends on states. During the original work-flow (Algorithm 2), the machine produces a symbol when state is entered. The canonical state is constructed to result in the label and output value $o_i \in \Delta$.

We propose to assign the attribute vector of integers (with length $|\Delta|$) to each state. The initial values for attributes are equal to 0.

Definition 2.35 (Attributed state) *qA consists of three parts:*

- *the $q.label$ part defines the ID of the state,*
- *the $q.value$ part describes the symbol that the machine outputs during the original work-flow,*
- *the $q.attributes: \; < o_1, o_2 \dots o_{m-1} >$ describes the vector of attributes with the length $|\Delta| = m$, wherein each attribute counts the number of times the corresponding output symbol was used.*

We present each Moore machine in the form $\mathrm{SR_S(MoFST)}$ (see Definition 2.32). Each $SR_S(MoFST.transition)$ corresponds to $\mathrm{DFA}_\emptyset$ (Theorem 2.10). Thus we can replace each state in $\mathrm{DFA}_\emptyset$ by attributed state $\forall q\{q \in Q : DFA_\emptyset\} \rightarrow qA$.

This $\mathrm{DFA}_\emptyset$ with attributed states can be transformed into a Moore machine by means of Algorithm 10, which consists of two parts:

- updating attributes with respect to the training set,
- setting up the output value $q.value$ according to attributes $q.attributes$.

Decorating $\mathrm{DFA}_\emptyset$ as a Moore machine

The decorating process is similar to the original machine run cycle (see Algorithm 2). The only difference lies in the processing of I/O pairs. During the original work-cycle, the Moore machine outputs symbols for each input string, while during the decorating cycle, the machine modifies inner attributes (see Algorithm 10).

The first part, i.e. updating attributes for each I/O pair in the training set, is done by the function $updateAttributes(input, output)$. The second part – translating attributes into output value – is done by the function, which assigns to the state value the output symbol that obtained the maximum attribute value.

Example 2.13 shows the basic transformations occurring inside the decoration algorithm.

Example 2.13 *Let us consider the Moore machine M_{aab} with 4 states $\Sigma = \{b, a\}$ and $\Delta = \{0, 1\}$ (see Example 2.10). Our training set is 'aab recognizer' (Figure 4.8). Figure 2.23 shows the process of decorating the M_{aab} represented by $SR_D(M_{aab}) = \{[0, 1, 0, 2, 3, 2, 0, 1]\}$.*

Each state $q \in \{q_0 \dots q_n - 1\}$ has the vector of attributes $< a_0, a_1 >$. Figures 2.23(a), 2.23(b), 2.23(c) show the process of updating the attribute

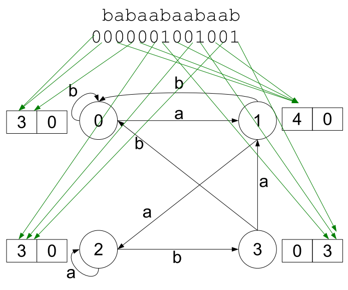
Algorithm 10 Decorating DFA$_\emptyset$ as a Moore machine

```
1:  for all I/O pairs do
2:      updateAttributes($input, output$)
3:  end for
4:  for $i = 0 \to n$ do
5:      $q_i.value \leftarrow \Delta[q_i.findMaxAttribute()]$
6:  end for
7:  return generated Moore machine

8:  function updateAttributes($iS, oS$)
9:      $q_c \leftarrow q_0$
10:     increment($q_c.attributes[\Delta[oS[0]_{position}]]$)
11:     for $i = 0 \to iS.size - 1$ do
12:         makeTransition($iS[i], oS[i+1]$))
13:     end for
14: end function

15: function makeTransition($ic, oc$)
16:     $transition \leftarrow TR[q_c.label][\Sigma[ic_{position}]]$
17:     if $\exists$ transition then
18:         $q_c \leftarrow transition.toState$
19:         increment($q_c.attributes[\Delta[oc_{position}]]$)
20:     end if
21: end function
```

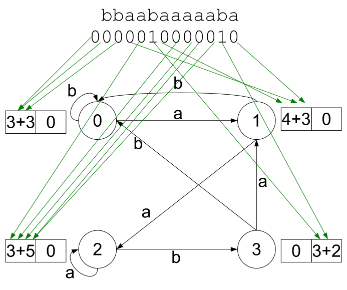
for three first I/O pairs. Each green arrow signifies the execution of $makeTransition(ic, oc)$ function. Figure 2.23(d) shows the process of assigning $q.value$ for each state.

Example 2.14 demonstrates how the objective function landscape changes if the method for deriving the output function is applied.

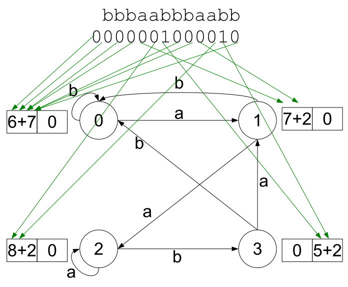
Example 2.14 *Suppose we need to construct a Moore machine, which is consistent with the training set 'ab Recognizer' (Figure 4.5). M_{ab} contains 3 states and the input and output alphabets are defined as $\Sigma = \{b, a\}$ and $\Delta = \{0, 1\}$. The objective function shows how accurately the current machine describes the training set. Figure 2.24(a) demonstrates the landscape of the objective function for one of the possible output functions (out of 2^3 possible output functions). The objective function value of a specific machine is marked on the vertical axis, while the horizontal axis presents all possible machines with an output function $SR_S(M_{ab}.output) = [0, 0, 1]$: $SR_S(M_{ab}) = \{\forall SR_S(M_{ab}.transition), [0, 0, 1]\}$. Figure 2.24(b) shows the landscape of the objective function for $SR_D(M_{ab}) = \{\forall SR_S(M_{ab}.transition)\}$.*



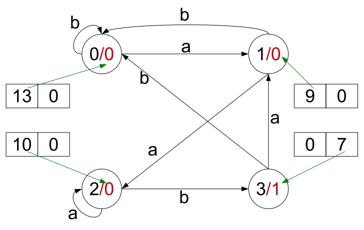
(a) I/O pair 1



(b) I/O pair 2

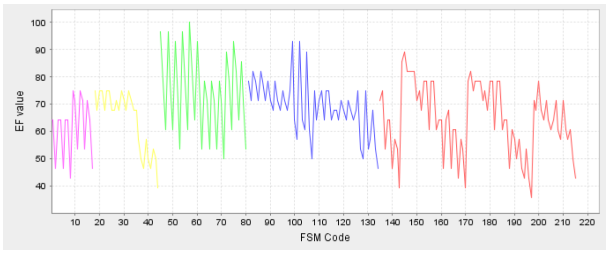


(c) I/O pair 3

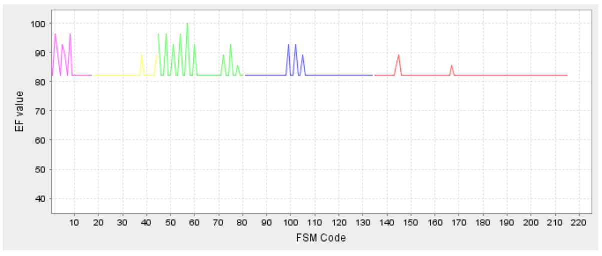


(d) Assigning values

Figure 2.23 Decorating $SR_D(M_{aab}) = [0, 1, 0, 2, 3, 2, 0, 1]$ *as a Moore machine*



(a) The output function is given $SR_S(M_{ab}.output) = (0, 0, 1)$



(b) The output function is derived $SR_D(M_{ab})$

Figure 2.24 'ab' Recognizer. Objective function landscape. Given output function vs. Derived output function

As these figures demonstrate, using the method based on deriving the output function results in a much flatter and less aggressive objective function landscape.

Deriving output function for other FSMs

As discussed in Subsection 2.4.2, deriving the output function can also be applied for other types of FSMs, for example a Mealy machine:

Definition 2.36 (Mealy machine with derived output)

$$SR_D(MeFST) = \{SR_D(MeFST.transition), MeFST.output\}$$

$\mathrm{SR_D(MeFST)}$ *representation is a special case of* $\mathrm{SR_S(MeFST)}$*, where*

- $SR_D(MeFST.transition) \leftarrow SR_S(MeFST.transition)$ *and*
- $MeFST.output$ *is constructed by algorithm.*

2.5. Problems with String Representations

The considered coding systems for FST are sensitive to the labeling order of the states. This leads to the situation, where the search space contains several machines which are isomorphic (Example 2.5). On the string representation level, it is not possible to state whether or not two represented FSTs are isomorphic. As regards the effectiveness of the search space algorithm this means duplicate points in search space, additional recalculation and complexity of the evaluation function landscape.

The number of such points (isomorphic FST) grows with the factorial of the number of states $(n-1)!$ (label '0' always belongs to initial state) due to the different possibilities of ordering the state labels. This problem is also mentioned in [31].

The idea is to construct a coding system, which automatically removes all isomorphic FSTs except one. This reduces the search space complexity.

Another problem with presented coding systems is the issue of unreachable states (Example 2.4). Although coding systems guarantee that for each string representation (except binary string representation) there is a corresponding FST, they do not guarantee that all states of the FST are reachable from the initial one. In the context of training data processing, this means that such a FST with unreachable states acts like a machine with fewer states. These unreachable states are redundant and can be removed. However, this can be problematic, especially if we define the problem statement as 'find FST with exactly n states'.

The points representing machines with unreachable states must be removed from the search space in order to reduce the space complexity and to guarantee that the searched machines have the required number of states.

2.6. Existing Solutions

Natalie Hammerman and Robert Goldberg (chapter from book [45]) presented a method which solves both issues of isomorphic machines as well as machines with unreachable states. It is based on constructing two algorithms, which rearrange the string representations of machines in order to ensure the removal of isomorphisms and machines with unreachable states:

- *SFS algorithm* (**S**tandardizing the transitions to the **F**uture or next **S**tates according to a mathematical function) – renames and reorders states which allows to remove isomorphisms;

- *MTF algorithm* (**M**oves the reachable states of a finite state machine **T**o the **F**ront of the genome) – solves the problem of unreachable states.

Both algorithms require a significant amount of computation, due to the recalculation of the string representations of the machines.

We propose a new coding system for presenting $[FST \to String]$, which solves the problem of unreachable states and isomorphisms on the representation level. Additional reordering algorithms are not required.

2.7. Canonical String Representation

The $SR_S(FSM.transition)$ code depends on the labels of the states and their ordering, as renaming states leads to isomorphisms. We can solve this problem by determining the algorithm by which the state labels are named. In order to do so, a method known as *normal form string* is adapted.

2.7.1. Preliminaries. Normal form strings

In order to describe our proposed string representation we consider the basic theory developed by Almeida, Moreira and Reis ([46], [39], [47]) in the context of enumerating of deterministic finite state acceptors (DFA). We present only specific definitions and algorithms that are required for our research. More information, proofs and other algorithms can be found in original sources.

The main idea of the approach by Almeida, Moreira and Reis is to find an unique string representation of initially connected DFA (ICDFA), i.e. all states are reachable from the initial one. This is done by constructing an ordering for state labels.

Suppose we have $ICDFA_\emptyset = (Q, \Sigma, \delta, q_0)$, where Q is a set of states $|Q| = n$, q_0 is initial states, Σ is the input alphabet with k symbols and δ is a transition function. The set of final states is omitted.

In order to construct the *canonical string representation* based on canonical order of the ICDFA states:

- the ordering of the input alphabet must be defined for a given $\Sigma = \{i_0, i_1, \ldots, i_{k-1}\}$, where the order is defined as $i_0 < i_1 < \ldots < i_{k-1}$. For example, lexicographical ordering can be used
- the set of states of a given ICDFA must be explored by using a breadth-first search by choosing the outgoing edges in the order of symbols in Σ.

Definition 2.37 (Normal form string) *For given a $ICDFA_\emptyset$, the representing string in the form $(s_j)_{j\in[0\ldots kn-1]}$ with $s_j = \delta(\lfloor j/k \rfloor, i_{j\ mod\ k})$ and $s_j \in [0\ldots n-1]$ satisfying rules:*

$$\begin{aligned}(\forall m \in [2\ldots n-1])(\forall j \in [0\ldots kn-1])\\ (s_j = m \Rightarrow (\exists l \in [0\ldots j-1])s_l = m-1)\end{aligned} \tag{2.7}$$

$$\begin{aligned}(\forall m \in [1\ldots n-1])\\ (\exists l \in [0\ldots km-1])s_l = m\end{aligned} \tag{2.8}$$

is called a normal form string [46].

There is the one-to-one mapping between *normal form string* $(s_j)_{j\in[0\ldots kn-1]}$ and non-isomorphic $ICDFA_\emptyset$ with n states and input alphabet Σ with k symbols.

Example 2.15 *Figure 2.6 shows two Moore machines that are isomorphic. If we choose ordering for Σ as $a < b$, the correct state label ordering is shown in Figure 2.6(b) and the corresponding canonical string for the transition function in this context is* $[0, 1, 2, 2, 0, 2]$. *This string covers two presented machines.*

Definition 2.38 (Flag) *In the canonical string representation $(s_j)_{j\in[0\ldots kn-1]}$, we can define flags $(f_j)_{j\in[1\ldots n-1]}$ that are a sequences of indexes of the first occurrence of state label j. The initial flag sequence is $(ki-1)_{i\in[1\ldots n-1]}$.*

The rules described before can be reformulated as

$$(\forall j \in [2\ldots n-1])(f_j > f_j - 1) \tag{2.9}$$

$$(\forall m \in [1\ldots n-1])(f_m < km) \tag{2.10}$$

For given k and n, the number of sequences $(f_j)_{j\in[1,n-1]}$, $F_{n,k}$ can be computed by

$$F_{k,n} = \binom{kn}{n}\frac{1}{(k-1)n+1} = C_n^{(k)}, \tag{2.11}$$

where $C_n^{(k)}$ are the Fuss-Catalan numbers. The proof can be found in [47].

The process of enumerating of all possible ICDFAs presented by the canonical string representation is presented in Algorithm 11, which consists of two parts:

1. generating flags presented by Algorithm 12 and
2. generating all sequences inside the flag described by Algorithm 13

and where n is defined as the number of states $|Q| = n$ and k is defined as the size of the alphabet $|\Sigma| = k$, in which the symbols are in a lexicographical order [47].

Algorithm 11 Canonical string representations enumerator $enumerate()$

```
Require: ICDFA number of states $|Q| = n$
Require: ICDFA size of input alphabet $|\Sigma| = k$
 1: $F_c \leftarrow initFlag()$
 2: while $F_c$ != lastFlag do
 3:     $F_c \leftarrow$ nextFlag($n - 1$)
 4:     $F[]$ add $F_c$;
 5: end while
 6: for all $F[]$ do
 7:     set $F_c$
 8:     $SEQ_c \leftarrow$ initSeq
 9:     while $SEQ_c$ != lastSeq do
10:         $SEQ_c \leftarrow$ nextICDFA$(()n - 1, k - 1)$
11:         $cSR[]$ add $SEQ_c$
12:     end while
13: end for
14: return $\forall cSR[]$
```

Algorithm 12 $nextFlag(i)$

```
 1: function nextFlag($i$)
 2:     if $i == 1$ then
 3:         $F_c[i] \leftarrow F_c[i] - 1$
 4:     else
 5:         if $F_c[i] - 1 == F_c[i - 1]$ then
 6:             $F_c[i] \leftarrow k \cdot i - 1$
 7:             nextFlag($i - 1$)
 8:         else
 9:             $F_c[i] \leftarrow F_c[i] - 1$
10:         end if
11:     end if
12: end function
```

Algorithm 13 $nextICDFA(a, b)$

```
function nextICDFA($a, b$)
    $i \leftarrow a \cdot k + b$
    if $a < n - 1$ then
        while $i \in F_c$ do
            $b \leftarrow b - 1$
            $i \leftarrow i - 1$
            if $b < 0$ then
                $b \leftarrow k - 1$
                $a \leftarrow a - 1$
            end if
        end while
    end if
    $F_j \leftarrow$ nearest flag not bigger than $i$
    if $SEQ_c[i] == SEQ_c[F_j]$ then
        $SEQ_c[i] \leftarrow 0$
        if $b == 0$ then
            nextICDFA($a - 1, k - 1$)
        else
            nextICDFA($a, b - 1$)
        end if
    else
        $SEQ_c[i] \leftarrow SEQ_c[i] + 1$
    end if
end function
```

2.7.2. $\mathrm{cSR_S(FST)}$: Canonical string representation of FST

We defined the string representation $\mathrm{SR_S(FST)}$ (Definitions 2.32, 2.33) and showed that $SR_S(FST.transition)$ corresponds to $\mathrm{DFA}_\emptyset$ (Theorems 2.8, 2.10).

The $SR_S(FST.transition)$ code depends on the labels of the states and their ordering, because renaming states leads to isomorphisms. We can solve this problem by determining the way the state labels are named.

There is a method for the enumeration of DFAs, which is based on a special coding system for representing the DFAs called the *canonical string representation*. It ensures that there are no isomorphisms and machines with unreachable states in the enumeration list (see Subsection 2.7.1).

We adapt this method in order to define the coding system for the FST. The algorithms and structure used for DFA enumeration can also be applied to representing the transition function of Moore machine. If we omit the output values at the states of the Moore machine, the resulting structure which represents only the transition function can be considered as a structure representing DFA without final states ($\mathrm{DFA}_\emptyset$). This allows us to apply methods of *normal form string* representation to Moore machine encoding.

The FST can be depicted by two strings, wherein one string stores information about the transition function, the second one stores the output values of the state. Thus, in order to apply the *canonical string representation* method we need to add ordering into the string that represents the transition function.

We define $\mathrm{cSR_S(FST)}$ as a special case of $\mathrm{SR_S(FST)}$ system (see Subsection 2.4.1), where $SR_S(FSM.transition)$ is represented by the *canonical string representation*.

Definition 2.39 *The separated canonical string representation of a Moore machine is a structure formed from* $\mathrm{SR_S(MoFST)}$*:*

$$cSR_S(MoFST) = \{cSR(MoFST.transition), SR_S(MoFST.output)\},$$

where $cSR(MoFST.transition)$ *is a canonical string representation of the Moore machine transition function.*

Definition 2.40 *The separated canonical string representation of a Mealy machine is a structure formed from* $\mathrm{SR_S(MeFST)}$*:*

$$cSR_S(MeFST) = \{cSR(MeFST.transition), SR_S(MeFST.output)\},$$

where $cSR(MeFST.transition)$ *is a canonical string representation of the Mealy machine transition function.*

Decoding algorithm for FST

In the case of heuristic search algorithms the set of possible solutions is generated in string form. Subsequently, we decode each string into FST for the evaluation process. Thus, the encoding process is usually not required.

To reconstruct a Moore machine from its string representation, the input and output alphabets and a pre-given number of states n must be defined. The decoding algorithm (Algorithm 14) works as follows:

- step 1 – create n states, label q_0 as the initial one and assign output values according to $SR_S(MoFST.output)$,
- step 2 – create a transition for each state, where 'from' state is the current state and 'to' state is the state labeled by an index found from $cSR(MoFST.transition)$.

2.7.3. $\mathrm{cSR_D(FST)}$: Canonical string representation

$\mathrm{cSR_D(FST)}$ is a special case of the $\mathrm{SR_D(FST)}$ system (see Subsection 2.4.2), where $cSR(FST.transition)$ is represented by the *canonical string representation* and the output function is derived from the training set.

Algorithm 14 $cSR_S(MoFST) \rightarrow MoFST$ transformation

```
Require: Input alphabet $\Sigma$ with $k$ symbols
Require: Input alphabet $\Delta$ with $m$ symbols
 1: for $i = 0 \rightarrow n - 1$ do
 2:     createState($q_i$)
 3:     $q_i.output \leftarrow \Delta[MoFST.output[i]]$
 4:     if $i == 0$ then
 5:         setInitial($q_i.$)
 6:     end if
 7: end for
 8: for $i = 0 \rightarrow n - 1$ do
 9:     $fromState \leftarrow q_i$
10:     for $j = 0 \rightarrow k - 1$ do
11:         $toState \leftarrow q_{CSR[i] \times k + j}$
12:         $inSymbol \leftarrow \Sigma[j]$
13:         $TR \leftarrow$ Transition($fromState, toState, inSymbol$)
14:     end for
15: end for
16: return Moore machine
```

Definition 2.41 *Canonical representation with a derived output is a structure formed from* $\mathrm{cSR_S(MoFST)}$*:*

$$cSR_D(MoFST) = \{cSR(MoFST.transition), [derived]\},$$

where $cSR(MoFST.transition)$ is a canonical string representation of the Moore machine transition function.

Definition 2.42 *Canonical representation with a derived output is a structure formed from* $\mathrm{cSR_S(MeFST)}$*:*

$$cSR_D(MeFST) = \{cSR(MeFST.transition), [derived]\},$$

where $cSR(MeFST.transition)$ is a canonical string representation of the Mealy machine transition function.

Both transformation algorithms $[cSR_D(MoFST) \rightarrow Moore\ machine]$ and $[cSR_D(MeFST) \rightarrow Mealy\ machine]$, are the same as for $[cSR_S(MoFST) \rightarrow Moore\ machine]$ (see Algorithm 14), although preliminary output function computation is required.

Further analysis of the $\mathrm{cSR_S(FST)}$ coding system and other algorithms, such as random generation of $\mathrm{cSR_S(FST)}$, can be found in Section 3.2.

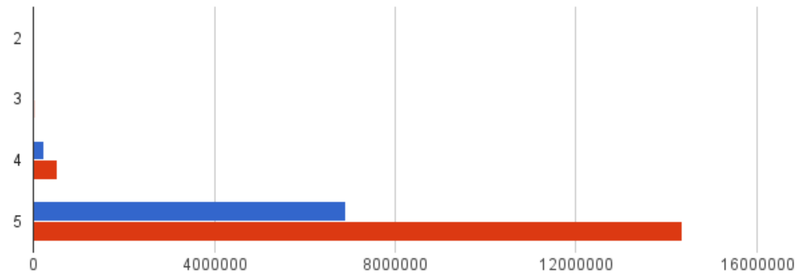
2.8. Space Complexity

Table 2.2 presents formulas for search space complexity in each coding system, if it's directly computable. Search space complexity shows the number of points that can be considered as FST in a search space for a given number of states n, the size of input alphabet k and the size of the output alphabet m.

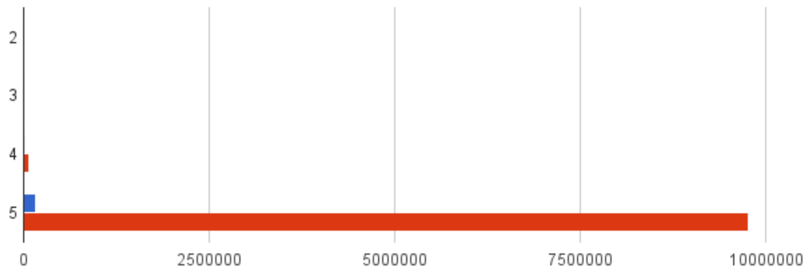
Table 2.2 $\mathrm{SR_S(FST)}$ *and* $\mathrm{SR_D(FST)}$. *Search space complexity*

	$\mathrm{SR_S(FST)}$	$\mathrm{SR_D(FST)}$
Moore machine	$\mathcal{O}((m \times n^k)^n)$	$\mathcal{O}(n^{nk})$
Mealy machine	$\mathcal{O}((m^k \times n^k)^n)$	$\mathcal{O}(n^{nk})$

In the $\mathrm{SR_D(FST)}$ coding system, the search space complexities for both cases (Moore and Mealy machines) are equal, because each search space only corresponds to the transition function, as output function is derived. Hence, the search space complexity is m independent.



(a) Search space complexity Vs. size of input alphabet, for given n=3



(b) Search space complexity Vs. number of states, for given k=2

Figure 2.25 Search space complexity dependency on k *and* n

For the $\mathrm{SR_S(FST)}$ coding system, wherein the output function itself is inside the search space, the search space of $\mathrm{SR_S(MeFST)}$ is much bigger than

$\mathrm{SR_S(MoFST)}$ for equal n, k and m. This can be explained by FST properties: the output function of a Moore machine is connected to states and output function of a Mealy machine is connected to transitions. Therefore, the size of the output function vector depends on the number of states n for a Moore machine and on the number of transitions $n \times k$ for a Mealy machine .

Figure 2.25 shows how the search space grows in relation to changes in the corresponding parameter:

- Figure 2.25(a) shows changes in search space size due to the change in k (the number of states $n = 3$). The red series is for the $\mathrm{SR_D(FST)}$ coding system (grows faster) and the blue series for the $\mathrm{cSR_D(FST)}$.

- Figure 2.25(b) shows changes in the search space size due to change in n (the number of symbols in the input alphabet $k = 2$). The red series is for the $\mathrm{SR_D(FST)}$ coding system which grows faster and the blue series is for $\mathrm{cSR_D(FST)}$.

Table 2.3 illustrates precomputed search space sizes for a given n, k, m in the case of a Moore machine.

Table 2.3 Moore machine. Search space complexity. Different coding systems

m	k	n	$\mathrm{cSR_D(FST)}$	$\mathrm{SR_D(FST)}$	$\mathrm{cSR_S(FST)}$	$\mathrm{SR_S(FST)}$
2	2	2	12	16	48	64
2	2	3	216	729	1728	5832
2	2	4	5248	65536	83968	1048576
2	2	5	160675	9765625	5141600	312500000
2	2	6	5931540	2176782336	379618560	139314069504
2	2	7	256182290	678223072849	32791333120	86812553324672
2	3	3	7965	19683	63720	157464
2	3	4	2128064	16777216	34049024	268435456
2	3	5	914929500	30517578125	2927774 4000	976562500000
2	4	2	240	256	960	1024
2	4	3	243000	531441	1944000	4251528
2	4	4	642959360	4294967296	10287349760	68719476736
2	5	2	992	1024	3968	4096
2	5	3	6903873	14348907	55230984	114791256

Table 2.4 illustrates precomputed search space sizes for a given n, k, m in the case of a Mealy machine.

Table 2.4 Mealy machine. Search space complexity. Different coding systems

m	k	n	$\mathrm{cSR_D(FST)}$	$\mathrm{SR_D(FST)}$	$\mathrm{cSR_S(FST)}$	$\mathrm{SR_S(FST)}$
2	2	2	12	16	192	256
2	2	3	216	729	13824	46656
2	2	4	5248	65536	1343488	16777216
2	2	5	160675	9765625	164531200	10000000000
2	2	6	5931540	2176782336	24295587840	8916100448256
2	2	7	256182290	678223072849	4197290639360	11112006825558000
2	3	3	7965	19683	4078080	10077696
2	3	4	2128064	16777216	8716550144	68719476736
2	3	5	914929500	30517578125	29980409856000	1000000000000000
2	4	2	240	256	61440	65536
2	4	3	243000	531441	995328000	2176782336
2	4	4	642959360	4294967296	42136984616960	281474976710656
2	5	2	992	1024	1015808	1048576
2	5	3	6903873	14348907	226226110464	470184984576

2.9. Conclusion

We discussed several representation of FST that can be used for the stochastic search algorithm and showed how FST can effectively be encoded into string of integers. We considered the problem of search space size and introduced a new system for string representation of FST. The proposed representation system divides the search space into non-intersecting parts. This allows to apply a heuristic search algorithm in parallel. The algorithms for random FSTs generation in string form and FSTs transformation were presented. The comparison between representation systems demonstrated that the proposed representation model is effective due to search space minimization.

3. SEARCH ALGORITHM

The *search problem* is defined as finding the optimum (minimum or maximum) of a given function. The set of *points*, which present the function arguments, provides the *search space*. For each point in the search space, there is a function value. The task is to find the point (argument), which gives the optimal function value.

If there is no information about the search space, then two main options remain:

1. either to exhaustively traverse the search space or,

2. to select certain random points and pick the most suitable one.

If the search space is infinite we define a *range* which restricts the search space. Traversing the entire search space is time-consuming, but provides exact results. In the case of infinite search space, the results also depend on how the search space range is defined. Moreover, the restricted area might not contain the optimum. Another approach is to generate a small amount of random points in the search space. This approach provides a fast solution, but its quality of the solution is questionable. The stochastic optimization method presents a compromise between an exhaustive search and choosing random points.

3.1. Preliminaries. Stochastic Optimization

Optimization techniques entail several approaches – *deterministic* procedures and *stochastic* procedures that contain randomness and probabilistic computations. The main advantage of *stochastic optimization* is that it can be applied to any search problem without specific knowledge about the structure of the search space. Stochastic optimization may also be helpful, when the complexity of deterministic methods grows rapidly in relation to the size of the search space.

There are two main properties that must be implemented in any stochastic optimization method – *exploration* and *exploitation* (see Figure 3.1). *Exploration* is an ability of the method to explore the entire search space in a global way and *exploitation* is the ability to focus on the local area and search for a more precise solution.

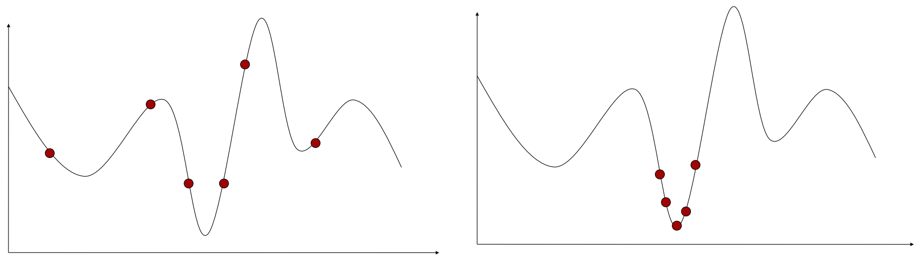

(a) Exploration (b) Exploitation

Figure 3.1 Exploration vs. exploitation

The family of stochastic optimization methods has several principles in common:

- The *search space* is defined as a set of points, where each point represents a *candidate solution*. Usually, candidate solutions are presented indirectly by some type of structure, which encodes the candidate solution. Initially, a certain fixed amount of points is generated randomly.
- The *score value*, which is assigned by using the *evaluation function* to show the quality of a solution.
- The search algorithm contains *modification operators* that allow to construct new solutions from existing ones.

Nature has inspired the construction of new stochastic search algorithms. There is a set of methods, such as *Evolutionary Algorithms*, *Genetic Algorithm*, *Evolutionary Programming*, which are based on the theory of evolution. Some methods simulate social behavior, for example *Particle Swarm Optimization (*PSO*)* imitates the social behavior of birds, while *Ant Colony Optimization* applies ideas related to the behavior of ants foraging for food. Some stochastic optimization methods are based on the laws of physics. For instance *Simulated Annealing* is based on thermodynamic effect, and *Central Force Optimization* and *Gravitational Search Algorithm* are based on gravitational force.

Central Force Optimization (CFO) is a deterministic gravity based search algorithm, which simulates a group of probes [48]. *Space Gravitational Optimization (SGO)* [49] simulates asteroids flying through a curved search space. A gravitationally-inspired variation of local search, *Gravitational Emulation Local Search Algorithm (GELS)* was proposed by Webster [50], [51]. The newest method, *Gravitational Search Algorithm (*GSA*)* was proposed by Rashedi [52], [53], [54] as a stochastic variation of CFO. A discrete modification of GSA was proposed by Zibanezhad [55] in the context of Web-Service composition.

3.1.1. Particle Swarm Optimization

Particle Swarm Optimization (PSO) algorithm is inspired by the social behavior of a specific set of objects, e.g. a flock of birds or school of fish [36]. The general idea behind this method involves a set of points in the search space, wherein each point has a value assigned by the *evaluation function*. The *task* is to find the optima of this function. In addition, there is a set of particles, that are moving in the defined search space. The movement laws can be considered as interactions between objects. Using movement laws objects can find the positions with better values or even optima.

Standard PSO algorithm

It is useful to consider the set of particles called a *swarm* in more detail. Each of these particles is characterized by a *position vector*, *velocity vector* and the best known position for this object. In addition, there is the global best known position for the whole swarm.

The *position* vector $p_d, d \in [0 \ldots n]$ presents a *candidate solution*. The dimensionality n of the vector depends on the problem size. We assign a numeric value for each candidate solution using the *evaluation function*. According to the assigned values we can choose the global best known position $Gbest$, which is the point with the optimal value found so far by the whole swarm, and the local best known position $Pbest$, which is the best position, that was found by this exact particle.

The *velocity* vector $v_d, d \in [0 \ldots n]$ represents the movement trend of the particle. It is computed by using the equation

$$v_d(t) = \alpha \cdot v_d(t-1) + \beta \cdot r_1 \cdot (Pbest_d - p_d(t-1)) + \gamma \cdot r_2 \cdot (Gbest_d - p_d(t-1)), \quad (3.1)$$

where:

- $\alpha,\ \beta,\ \gamma$ are learning coefficients with α representing the inertia, β the cognitive memory and γ the social memory. Learning coefficients must be defined by the user.
- r_1 and r_2 are random values in range $[0 \ldots 1]$.
- $Pbest_d$ is the local best known position for the particle, $Gbest_d$ is the global best known position of the swarm.
- $v_d(t)$ is the new value of the velocity vector at dimension d and $v_d(t-1)$ is the previous velocity value.

The new position $p_d(t)$ is simply defined as the sum of the previous position $p_d(t-1)$ and the new velocity $v_d(t)$:

$$p_d(t) = p_d(t-1) + v_d(t). \quad (3.2)$$

The PSO algorithm is presented in Algorithm 15. The initialization part consists of defining the required learning parameters, establishing the search space boundaries and generating the swarm with a random position and velocity. The search process can be described as an iterative updating of the positions and velocities. The process ends when the ending criteria are met – either the number of iterations is exceeded or the optimal solution is found. The evaluation function f and the dimensionality of vectors are problem specific.

Algorithm 15 Standard Particle Swarm Optimization

```
1: setBounds($B_{up}$, $B_{low}$)
2: setLearningCoefficients($\alpha$, $\beta$, $\gamma$)
3: defineSwarmSize($s$), setIterations($e$)
4: for $i = 0 \to s - 1$ do
5:     initParticlePosition($p^i$, $B_{up}$, $B_{low}$)
6:     $Pbest^i \leftarrow p^i$
7:     initializeParticleVelocity($v^i$, $B_{up}$, $B_{low}$)
8:     evaluateParticle($f(p^i)$)
9:     if $f(Gbest^i) < f(p^i)$ then
10:        $Gbest^i \leftarrow p^i$
11:    end if
12: end for
13: while current iteration $< e$ and optimal solution is not found do
14:    for $i = 0 \to s - 1$ do
15:        updateVelocity($v^i$) for each dimension (see Equation 3.1)
16:        updatePosition($p^i$) for each dimension (see Equation 3.2)
17:        if $f(Pbest^i) < f(p^i)$ then
18:            $Pbest^i \leftarrow p^i$
19:            if $f(Gbest^i) < f(p^i)$ then
20:                $Gbest^i \leftarrow p^i$
21:            end if
22:        end if
23:    end for
24: end while
```

Problems with the standard PSO algorithm

1. In Algorithm 15, there are three learning coefficients α, β, γ for adjusting the convergence abilities of the algorithm. Learning coefficients must be defined by the user according to the problem statement. One of the chief problems with this algorithm is the lack of deterministic methods for finding values of learning coefficients. However, there are several non-deterministic methods for solving this problem. For example, in the case of empirical methods we can try several parameter values and observe the behavior of the PSO algorithm in order to choose the best ones. In the case of meta-heuristics, the choice of parameter values can also be

considered as a search problem. Hence, heuristic optimization can also be applied.

2. The second problem lies in defining of Equation 3.1. As is evident, there is a $Gbest$ position, which is valid for the whole swarm and does not take into account the distance between the particle and the global best position $Gbest$. In some situations, if $Gbest$ itself is in a bad zone, then the whole swarm falls into a local optima.

   There are two main solutions for the second problem:

   - either defining the *neighborhood* of every particle by taking into account the $Gbest$ for the group of particles, which are close to each other, rather than the $Gbest$ for the whole swarm, or
   - defining *parallel swarms*, where groups of particles move in the search space without any interactions between the groups.

3.1.2. Gravity as inspiration for optimization algorithms

There are four main forces acting in our universe, namely – gravitational, electromagnetic, weak-nuclear and strong nuclear. These main forces define the way our universe behaves and appears. The weakest force is gravitational, which defines how objects move depending on their mass.

The gravitational force between two objects i and j is directly proportional to the product of their masses and inversely proportional to the square distance between them

$$F_{ij} = G\frac{M_j \cdot M_i}{R_{ij}^2}. \tag{3.3}$$

Knowing the force acting on a body, we can compute acceleration as

$$a_i = \frac{F_i}{M_i}. \tag{3.4}$$

The search algorithms based on gravity adapt the following ideas:

- Each object in the universe has a mass and position.
- There are interactions between objects, which can be described using the law of gravity.
- Bigger objects (with a greater mass) create larger gravitational fields and attract smaller objects.

During the last decade, some researchers have tried to adapt the idea of gravity to discover optimal search algorithms. Such gravity based search algorithms have certain general features in common:

- The system is modeled on objects with a mass.
- The position of the objects describes the solution and the mass of the objects depends on the evaluation function.
- Objects interact with each other using the gravitational force.
- Objects with a greater mass present the points in the search space that have a better solution.

Using these characteristics, it is possible to define the family of optimization algorithms based on gravitational force. For example, *Central Force Optimization (CFO)* is a deterministic gravity based search algorithm proposed and developed by Formato [48]. It simulates the group of probes which fly into search space and explore it. Another algorithm, *Space Gravitational Optimization (SGO)* was developed by Hsiao and Chuang [49] in 2005. It simulates asteroids flying through a curved search space. A gravitationally-inspired variation of the local search algorithm, *Gravitational Emulation Local Search Algorithm (GELS)* was proposed by Webster [50], [51]. The newest method, *Gravitational Search Algorithm (*GSA*)* was proposed by Rashedi [53] as a stochastic variation of CFO.

In essence, the gravitationally-inspired algorithms are quite similar to PSO algorithms. Instead of a particle swarm, there is a set of bodies with masses. Moreover, the ideas of position and velocity vectors are the same, and the movement laws are also similar.

Our idea is to combine the PSO algorithm with the gravitationally-inspired search algorithm in order to produce a superior one.

3.1.3. Gravitational Search Algorithm

Gravitational Search Algorithm (GSA) was proposed by Rashedi as a stochastic variation of *CFO* to solve the problem of dependency on the generation of an initial population. The main difference between the GSA compared to the the CFO lies in randomness of some movements, which add an exploration factor to the algorithm.

A system of N objects, each of which is described by a real-valued position vector:

$$X_i = \left(x_i^1, \ldots, x_i^d, \ldots, x_i^n\right), d = [1 \ldots n]; \tag{3.5}$$

There x_i^d represents the position on ith object in dimension d.

At the specific time t we can recompute the forces, which are applied to the object i with a mass M_i by an agent j with a mass M_j:

$$F_{ij}^d(t) = G(t)\frac{M_{pi}(t) \cdot M_{aj}(t)}{R_{ij} + \epsilon}(x_j^d - x_i^d). \tag{3.6}$$

Here

$$G(t)\frac{M_{pi}(t)\cdot M_{aj}(t)}{R_{ij}+\epsilon} \tag{3.7}$$

corresponds to (3.3) and ϵ is a parameter and the part $(x_j^d - x_i^d)$ is required for computing the vector of the coordinates, if the coordinates of the beginning and end points are known.

R_{ij} is the Euclidean distance between two agents

$$R_{ij} = \|X_i(t), X_j(t)\|\,. \tag{3.8}$$

The gravitational constant is computed by:

$$G(t) = G(G_0, t). \tag{3.9}$$

In physics the general force acting on an agent must be computed as the vector sum of all acting points. The authors of the GSA algorithm proposed to add a stochastic characteristic to algorithm. Thus the general force is computed as:

$$F_i^d(t) = \sum_{j=1, i\neq j}^{N} rand_j F_{ij}^d(t), rand \in [0\dots 1]. \tag{3.10}$$

The acceleration of agent i can be computed based on knowing its inertial mass M_{ii} and force $F_i^d(t)$:

$$a_i^d(t) = \frac{F_i^d(t)}{M_{ii}(t)}; \tag{3.11}$$

Knowing the current acceleration, velocity and position can be recomputed:

$$v_i^d(t+1) = rand_i v_i^d(t) + a_i^d(t), rand \in [0\dots 1]. \tag{3.12}$$

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1). \tag{3.13}$$

The masses of agents are calculated from the calculated quality measure. A heavier mass means that the quality of that object is better. This agent has a bigger attraction and inertia, i.e. it moves slowly toward other agents. The quality measure is calculated using the fitness function fit_i. The masses are calculated as follows:

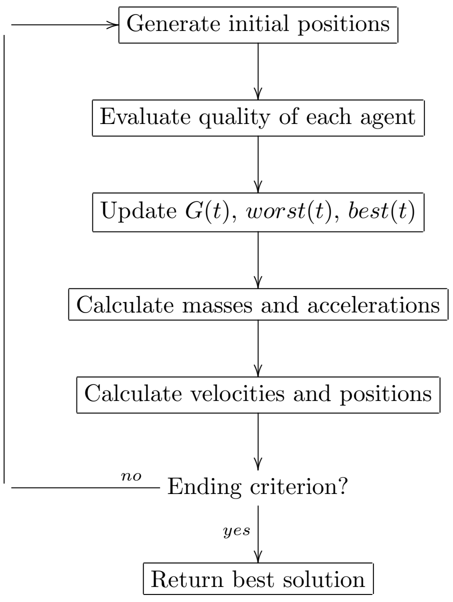
$$M_{ai} = M_{pi} = M_{ii} = M_{ii}, i = [1, 2, \dots N], \tag{3.14}$$

$$m_i = \frac{fit_i(t) - worst(t)}{best(t) - worst(t)}, \tag{3.15}$$

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}, \tag{3.16}$$

where the $worst(t)$ and the $best(t)$ are defined for the maximization problem as:

$$best(t) = \underbrace{min}_{j \in [1...N]} \; fit_j(t); \tag{3.17}$$



Generate initial positions
Evaluate quality of each agent
Update $G(t)$, $worst(t)$, $best(t)$
Calculate masses and accelerations
Calculate velocities and positions
Ending criterion?
no
yes
Return best solution

Figure 3.2 General procedure of GSA

3.2. The Search Space

According to our problem statement, the search space is a set of points representing a FST (Mealy or Moore machine) with n states, an input alphabet Σ with k symbols and an output alphabet Δ with m symbols. For each point, a score value is computed using one of the objective functions. This chapter presents the search algorithm, wherein the search space is defined by a set of FST, which are presented as

- $\mathrm{cSR_D(FST)}$s (see Subsection 2.7.3), if the output function can be restored from training data
- $\mathrm{cSR_S(FST)}$s (see Subsection 2.7.2), if the output of the machine cannot be inferred and must be searched.

The search algorithm consists of two stages:

- **first**: the search space is subdivided to subspaces, a score value is assigned for each subspace and the subspace with the best score value is returned,
- **second**: a more detailed search in the chosen subspace is carried out, and if a solution is not found, the next subspace with a higher score is searched.

There are several string representations of FST (see Chapter 2). However, according to our analysis of search space complexities (see Section 2.8), it is more reasonable to use the canonical string representation of FST ($\mathrm{cSR_D(FST)}$ or $\mathrm{cSR_S(FST)}$).

Two situations are considered. In the first one, the output function of the FST can be reconstructed from the training set ($\mathrm{cSR_D(FST)}$) and the search algorithm is only required for searching the FST structure. In the second situation, the structure of the FST and its output function must be searched (FST is represented by string $\mathrm{cSR_S(FST)}$). However, these two cases are similar as the string representation $\mathrm{cSR_D(FST)}$ is a special case of the string representation $\mathrm{cSR_S(FST)}$.

The different types of FSMs are represented differently. We propose the following representations for the Moore and Mealy machines (see Subsections 2.7.2, 2.7.3):

- $cSR_D(MoFST) = \{cSR[MoFST.transition],\ [derived]\}$,
- $cSR_S(MoFST) = \{cSR[MoFST.transition],\ SR[MoFST.output]\}$,
- $cSR_D(MeFST) = \{cSR[MeFST.transition],\ [derived]\}$,
- $cSR_S(MeFST) = \{cSR[MeFST.transition],\ SR[MoFST.output]\}$.

3.2.1. Search space structure

In fact, the transition function part of the FST – $SR[FST.transition]$ is a $\mathrm{FA}_\emptyset$ (Theorem 2.8 and Theorem 2.10) and can be processed in a similar way as $\mathrm{FA}_\emptyset$.

According to the theory of *normal form strings* (see Subsection 2.7.1), the canonical string representations of FA have the following properties: each $\mathrm{FA}_\emptyset$ is defined only by a single canonical string representation (no isomorphisms) and the set of all possible $\mathrm{FA}_\emptyset$ can be subdivided into subsets identified by *flags* (see Definition 2.38).

We adapt the theory of *normal form strings* to describe the properties of canonical string representation of FST and the search space.

Algorithm 11, which describes the enumeration of all possible $\mathrm{DFA}_\emptyset$ consists of two parts – the generation of all flags (see Definition 2.38) and the generation of all possible sequences in a given flag. In our context, this information can be used for subdividing the whole search space into non-intersecting subspaces, where each subspace is characterized by a corresponding flag (Figure 3.3).

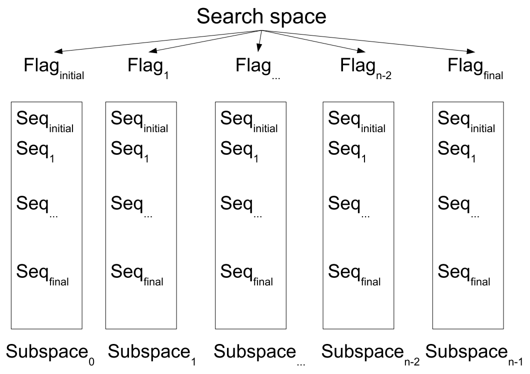


Figure 3.3 Search space structure

Definition 3.1 (Universe) *The universe is a set of all possible* FST*s, represented by canonical string representation (*$\mathrm{cSR_D(FST)}$ *or* $\mathrm{cSR_S(FST)}$*), where canonical string representations of transition functions belong to one flag.*

Definition 3.2 (Multiverse) *The multiverse is a set of all possible universes defined by flags.*

The multiverse defines the set of subspaces described by corresponding flags, and the universe is a subspace in itself and contains the points representing FSTs.

Example 3.1 *This example illustrates the definitions of multiverse and universes for the search space of* $\mathrm{cSR_D(FST)}$ *with 4 states and 2 symbols in its input alphabet.*

According to Equation 2.11, this multiverse has 14 universes. The description of such a search space is provided in Figure 3.4, where each row presents one universe and the 'size' shows the number of possible FST*s in that universe.*

3.2.2. Size of the multiverse

In order to optimize the search algorithm we need to know the size of the search space. For answering the question of 'how big the search space is', the following problems must be solved:

1. How many universes are there in the Multiverse, i.e. is how many flags that define subspaces are there?
2. How big is the universe, i.e. how many corresponding sequences can be generated for a given flag?
3. How big is a set of possible output functions (in the case of $\mathrm{cSR_S(FST)}$ representations) for a Moore machine and for a Mealy machine separately?

```
Size of search space: 5248
 0: (1, 3, 5): * 1 * 2 * 3 * *  Size: 96 From: 0 to: 95
 1: (1, 3, 4): * 1 * 2 3 * * *  Size: 128 From: 96 to: 223
 2: (1, 2, 5): * 1 2 * * 3 * *  Size: 144 From: 224 to: 367
 3: (1, 2, 4): * 1 2 * 3 * * *  Size: 192 From: 368 to: 559
 4: (1, 2, 3): * 1 2 3 * * * *  Size: 256 From: 560 to: 815
 5: (0, 3, 5): 1 * * 2 * 3 * *  Size: 192 From: 816 to: 1007
 6: (0, 3, 4): 1 * * 2 3 * * *  Size: 256 From: 1008 to: 1263
 7: (0, 2, 5): 1 * 2 * * 3 * *  Size: 288 From: 1264 to: 1551
 8: (0, 2, 4): 1 * 2 * 3 * * *  Size: 384 From: 1552 to: 1935
 9: (0, 2, 3): 1 * 2 3 * * * *  Size: 512 From: 1936 to: 2447
 10: (0, 1, 5): 1 2 * * * 3 * *  Size: 432 From: 2448 to: 2879
 11: (0, 1, 4): 1 2 * * 3 * * *  Size: 576 From: 2880 to: 3455
 12: (0, 1, 3): 1 2 * 3 * * * *  Size: 768 From: 3456 to: 4223
 13: (0, 1, 2): 1 2 3 * * * * *  Size: 1024 From: 4224 to: 5247
```

Figure 3.4 All subspaces for the $\mathrm{cSR_D(FST)}$ *search space (4 states and 2 symbols in input alphabet)*

Number of universes

The number of universes, which corresponds to the number of flags, can be computed by Equation (2.11):

$$F_{k,n} = \binom{kn}{n}\frac{1}{(k-1)n+1} = C_n^{(k)},$$

Size of the universe

The size of the universe is the number of sequences, which correspond to a pattern defined by a flag.

Algorithm 16 Computing universe size

```
Require: $FLAG[]$ characterizing universe
  $finalSEQ[] \leftarrow$ generateFinalSEQ($FLAG[]$)
  for $i = 0 \rightarrow finalSEQ.size - 1$ do
      if $finalSEQ[i] > 0$ then
          $size \leftarrow size \times (finalSEQ[i] + 1)$
      end if
  end for
  for $i = 0 \rightarrow n$ do
      $size \leftarrow size/(i + 1)$
  end for
  return size
```

The method for counting the number of sequences is presented in Algorithm 16. Example 3.2 demonstrates how to count the size of the universe defined by the flag $(1, 3, 4)$ using Algorithm 16.

Example 3.2 *The flag* $(1, 3, 4)$ *defines the sequence pattern* $[* \ 1 \ * \ 2 \ * \ 3 \ * \ *]$ *and the final sequence in this pattern is* $[0 \ 1 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3]$. *We can compute the number of allowed characters in a specific place by incrementing the numbers corresponding to those places (except flag defining characters):* $[1 \ - \ 2 \ - \ 3 \ - \ 4 \ 4]$. *The size of the subspace is multiplication of those numbers (see Figure 3.5).*

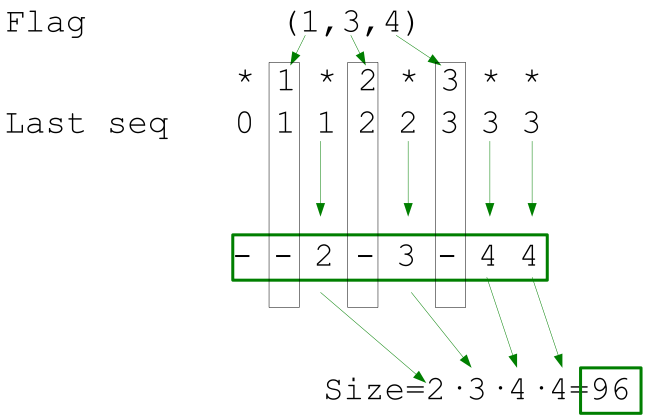


Figure 3.5 Counting the size of the universe defined by the flag $(1, 3, 4)$

Table 3.1 presents the following numerical information, where n is a given number of states and k number of characters in the input alphabet:

- The size of a flag set is computed by Equation (2.11),
- The biggest universe is the universe defined by the last flag (due to the specifics of the flag generation order) and its size is computed by Algorithm 16.

Number of output functions

In the case of $\mathrm{cSR_D(FST)}$ representation, only the number of possible transition functions are taken into account. There is only 'one dimension' – the structure of the machine presented by the canonical string, because each FST is represented by one vector.

For the $\mathrm{cSR_S(FST)}$ representation the search space grows, because we add 'the second dimension' – the string representation of the output function.

Table 3.1 Number of universes and size of the biggest universe

k	n	Max subspace size	Number of flags
2	2	8	2
2	3	81	5
2	4	1024	14
2	5	15625	42
2	6	279936	132
2	7	5764801	429
3	3	2187	12
3	4	262144	55
3	5	48828125	273
4	2	128	4
4	3	59049	22
4	4	67108864	140
5	2	512	5
5	3	1594323	35

Therefore, each FST structure can have several output functions (Figure 3.6). Moreover, each FST is represented by two vectors.

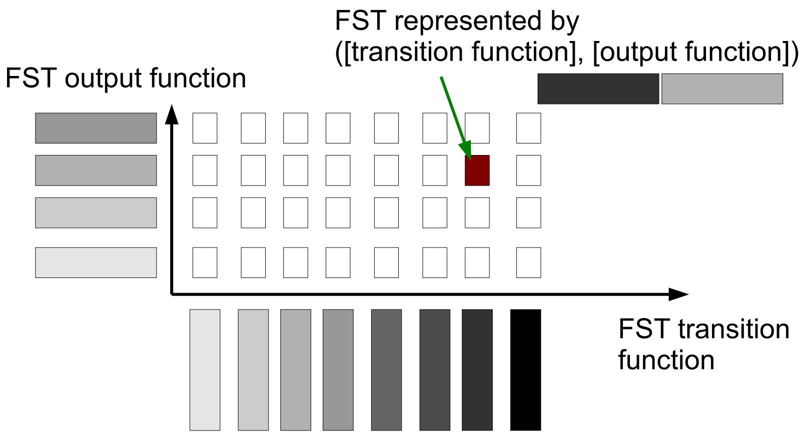


Figure 3.6 Two dimensional representation of FST

The length of the output function representation and the number of output functions depends on the machine type (n is the number of states, k is the size of input alphabet, m is the size of output alphabet).

- For a **Moore machine** the length of the string representation of the output function is n and the number of possible output functions is m^n,

- For a **Mealy machine** the length of the string representation of the output function is $n \times k$ and equals to the length of the transition function representation. The number of possible output function is m^{nk}.

For the situation with $\mathrm{cSR_D(FST)}$ representation, the length of the output function representation is defined as 1, and the number of all possible output functions is defined by $[0]$. Thus, the representation $\mathrm{cSR_D(FST)}$ can also be presented by $cSR_S(FST) = \{cSR[FST.transition], \ [0]\}$.

3.2.3. Storing points in universe

Search space contains points, which represent FSMs with score values. Universe is defined as a set of such points. It is useful to store point values in the following cases:

- **Visualization**. Storing values of points is useful during the search space exploration process, as it allows to demonstrate the landscape of the search space (see Section 3.2.7). This helps to illustrate the process of the search algorithm and to make required adjustments for to the search algorithm.
- **Performance improvement**. During the search process, the objective function is frequently recalculated. Sometimes calculating the score value is expensive, so a better idea is to store points with already calculated values, which improves the performance of the search algorithm [56], [57].

We propose to store point values in an associative array. The associative array (dictionary) is an abstract structure, which allows to store the values associated with the key (see Figure 3.7).

key_0	$value_0$
key_1	$value_1$
$key_{...}$	$value_{...}$
key_n	$value_n$

Figure 3.7 Associative array

The main operations allowed with an associated array are: $put(key, value)$, $remove(value)$, $containsKey(key)$, $containsObject(value)$, $get(key)$.

Although, an associative array is an abstract structure and its effective implementation is problematic itself, for example Java language contains the following implementations:

- $HashMap$ – the implementation of an associative array structure using hash tables,

- $TreeMap$ – the implementation of associative array structure using Red-Black tree.

The key must be unique, so specially constructed key must be introduced for our purposes.

Definition 3.3 (Key) *is a triple $key =< key.u,\ key.t,\ key.o >$, which is unique and is designed as triple of integers, wherein:*

- *$k.u$ – unique integer, representing the universe id (see Subsection 3.2.1),*
- *$k.t$ – unique integer, representing the transition function (see Algorithm 17),*
- *$k.o$ – unique integer, representing the output function (see Algorithm 18).*

cSR(FST.transition) to key.t transformation

For each $cSR(FST.transition)$, we can find an integer which is unique. This integer corresponds to the number in the generation order. We separate the process of sequence generation into two phases: the generation of all possible flags (defining subspaces) and the generation of sequences inside the subspace.

In order to find the integer corresponding to $cSR(FST.transition)$ we first of all find the local number in the generation order and if we want to obtain the number in the global ordering $subspace.iRange$ needs to be added.

Example 3.3 shows the process of transforming $[CSR(FST.transition) \rightarrow key.t]$.

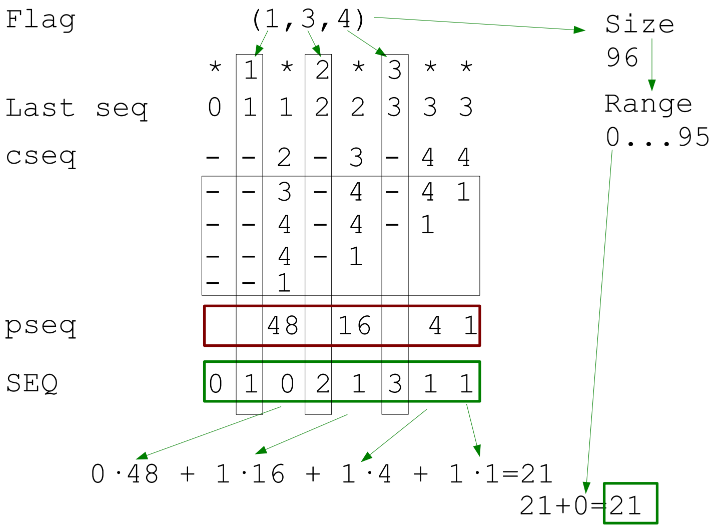


Figure 3.8 cSR(FST.transition) to key.t

For the transformation $[sequence \to integer]$ in a universe we need to predefine 'mask' based on the flag, describing the local subspace (function $generatePSeq(flag[])$). This pre-computation is only done once, when defining the universes (see Algorithm 17).

Example 3.3 *Transforming* $cSR_D(FST) : [0\ 1\ 0\ 2\ 1\ 3\ 1\ 1] \to 21$*:*

- *sequence* $[0\ 1\ 0\ 2\ 1\ 3\ 1\ 1]$ *belongs to subspace defined by flag* $(1, 3, 4)$*,*
- *the mask is* $[-\ -\ 48\ -\ 16\ -\ 4\ 1]$.

Figure 3.8 depicts the entire computation process.

SR(FST.output) to key.o transformation

For each output function of FST a unique integer number can be constructed that corresponds with the number in the generation order. This process is easier than the $[CRS(FST.transition) \to key.t]$ transformation, because no sequences are omitted here. Algorithm 18 shows the process of the $key.o$ computation.

For each FST, we know the length of the output function: n for a Moore machine and $n \times k$ for a Mealy machine as well as the size of the output alphabet m. Therefore, we can recompute the number in the generation order. The enumeration process here is similar to generating all possible numbers in a m-base numeric system, thus making the coding process similar to the transformation [m-base number $\to$ decimal number].

Algorithm 18 $SR(FST.output) \to key.o$ transformation

```
1: $code = 0$
2: $base = m$
3: for $i = SR(FST.output).length - 1 \to 0$ do
4:     $coef = base^{SR(FST.output).length-1-i}$
5:     $code+ = SR(FST.output)[i] \times coef$
6: end for
7: return $code$
```

Example 3.4 *Suppose we have a Moore machine with* $n = 4$ *states,* $k = 2$ *symbols in input alphabet* $\Sigma = \{a, b\}$ *and* $m = 2$ *symbols in output alphabet* $\Delta = \{0, 1\}$*. The length of* $SR(FST.output)$ *is 4, the number of all possible output functions* $2^4 = 16$ *(e.g. see Figure 2.20). For each output function, we can find the integer number* $key.o \in [0 \dots 15]$.

For example, for $SR(FST.output) = [0, 1, 1, 0]$ *the* $key.o = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$ *(according to Algorithm 18).*

Algorithm 17 $FST.transition \rightarrow key.t$: $transform(Seq[])$

```
Require: $subspace.flag[]$
Require: $subspace.iRange$
 1: function transform($Seq[]$)
 2:     $subspace.pSeq[] \leftarrow$ generatePSeq($subspace.flag[]$)
 3:     for $i = 0 \rightarrow subspace.pSeq.size - 1$ do
 4:         $code \leftarrow code + Seq[i] \times subspace.pSeq[i]$
 5:     end for
 6:     $code \leftarrow code + subspace.iRange$
 7: end function

 8: function generatePSeq($flag[]$)
 9:     $lastSeq \leftarrow$ genLastSeq($flag[]$)
10:     for $i = 0 \rightarrow lastSeq.size - 1$ do
11:         $cSeq[i] \leftarrow lastSeq[i] + 1$
12:     end for
13:     for $i = 0 \rightarrow flag[1] - 1$ do
14:         $cSeq[i] \leftarrow 0$
15:     end for
16:     for $i = 1 \rightarrow flag[].length - 1$ do
17:         $cSeq[flag[i]] \leftarrow 0$
18:     end for
19:     $pSeq[lastSeq.size - 1] \leftarrow 1$
20:     for $i = lastSeq.size - 2 \rightarrow flag[1] + 1$ do
21:         $c \leftarrow 1$
22:         for $j = i + 1 \rightarrow lastSeq.size - 1$ do
23:             if $cSeq[j] > 0$ then
24:                 $c \leftarrow c \times cSeq[j]$
25:             end if
26:         end for
27:         $pSeq[i] \leftarrow c$
28:     end for
29:     for $i = 1 \rightarrow flag[].length - 1$ do
30:         $pSeq[flag[i]] \leftarrow 0$
31:     end for
32: end function
```

3.2.4. Initialization

Before the search process can commence, certain initial computations are required. First of all, we need to define several preliminary parameters such as:

- FSM **type** – a problem-specific parameter defined in the task description.
- **Number of states** n – also problem-specific, but can be modified by the user and depends on the size and complexity of the problem.
- **Size of input alphabet** k**.** The input alphabet itself is not necessary at the search phase, because the search algorithm works on string representation level, but it is required on the level of $[SR \to FSM]$ transformation.
- **Size of output alphabet** m**.** The output alphabet depends on the search problem. As with the input alphabet, it is not required during the search phase, only the size of the alphabet matters.
- $getScore$ **function** required for evaluating FSM. So that the search algorithm sends the FSM representation to the environment through the encoder and as a result a score value in the range $[0 \ldots 1]$ is obtained. For the maximization task, an FSM with value 1 is the best one.

Secondly, based on the predefined problem-specific parameters we constructed, the following information is required for search space initialization:

- **Number of universes in a multiverse.** Based on n and k, the number of non-intersecting subsets in the search space can be calculated (see Subsection3.2.2, Equation (2.11).
- **Universes are defined by flags.** Using Algorithm 12 and parameters n and k, all possible flags from initial to final ones are generated. Further, a corresponding universe is created for each flag.
- **Length of** $SR(FST.output)$**.** Based on parameter m and machine type, we can compute the length of $SR(FST.output)$ and the number of all possible $SR(FST.output)$ (see Subsection 3.2.2). If $m = 0$, then we assume that the output function is derived from the training set.
- **Size of the universe.** For each universe we compute the number of all possible $cSR(FST.transition)$ with Algorithm 16. If $m > 0$, then it is assumed that the number of points in the universe is equal to the multiplication $|cSR(FST.transition)| \times |SR(FST.output)|$.
- **Mask for** $cSR(FST.transition) \to key.t$**.** Based on the flag, which describes the universe, we generate the mask for the transformation $cSR(FST.transition) \to key.t$ by function $generatePSeq$ (see Algorithm 17).

- **The number of first points in the universe.** Due to the sequential initialization of universes we can compute the code of the first point in the universe, if the size of previous universe is known.

The whole process of multiverse initialization is presented by Algorithm 19.

Algorithm 19 Define Multiverse

```
1: $Multiverse.type \leftarrow FSM.type$
2: $Multiverse.n \leftarrow number\ of\ states$
3: $Multiverse.k \leftarrow |\Sigma|$
4: $Multiverse.m \leftarrow |\Delta|$
5: $Multiverse.size \leftarrow number\ of\ flags$
6: $Multiverse.SRoutputLength \leftarrow$length($SR(FST.output)$)
7: $FLAGS[] \leftarrow$generaleAllFlags
8: $iRange \leftarrow 0$
9: for all $FLAGS[]$ do
10:     createUniverse($FLAG$)
11:     $Universe.flag \leftarrow FLAG$
12:     $Universe.iRange \leftarrow iRange$
13:     $Universe.size \leftarrow$countSize($FLAG$)
14:     $Universe.mask \leftarrow$ createFSTtoIntegerMask($FLAG$)
15:     $iRange \leftarrow iRange + subspace.size$
16: end for
```

3.2.5. Generating random FST

As described above, a FST can be defined by two dimensions – $cSR(FST.transition)$ and $SR(FST.output)$, when the FST is encoded by $cSR_S(FST)$. One dimension $cSR(FST.transition)$ is used, when the FST is presented by $cSR_D(FST)$, while the second dimension $SR(FST.output)$ is fixed to $\{[0]\}$). Thus, in order to generate random FST, we need to separately generate random $cSR(FST.transition)$ and random $SR(FST.output)$ (in the case of a $\mathrm{cSR_S(FST)}$ string representation).

Generating random cSR(FST.transition)

The algorithm for the $cSR(FST.transition)$ generation part is similar to the algorithm for uniform random generation of ICDFA (see [47]). Although we need to modify this algorithm, because we are only interested in generation of points in a subspace (universe), rather than entire search space.

Algorithm 20 $cSR(FST.transition)$ random generator in a universe

```
Require: $FLAG[]$ characterizing universe
 1: for $i = 0 \to rndCSR[].size$ do
 2:     $rndCSR[i] \leftarrow -1$
 3: end for
 4: for $j = 1 \to FLAG[].size - 1$ do
 5:     $rndCSR[FLAG[j]] \leftarrow j$
 6: end for
 7: $fi \leftarrow FLAG[].size - 1$
 8: $si \leftarrow rndCSR[].size - 1$
 9: while $si > -1$ do
10:     if $rndCSR[si]! = fi$ then
11:         $rndCSR[si] \leftarrow$ random($fi + 1$)
12:         $si - -$
13:     else
14:         $fi - -$
15:         $si - -$
16:     end if
17: end while
18: return $rndCSR[]$
```

Taking into account the knowledge that the search space is subdivided into non-intersecting subspaces (universes) characterized by flags we define the algorithm for generating a random string representation $cSR(FST.transition)$ in the local universe (Algorithm 20).

Generating random SR(FST.output)

The algorithm for generating a random $SR(FST.output)$ is simpler than $cSR(FST.transition)$ because there is need to omit certain points and the enumeration of all possible combinations is sequential.

The $SR(FST.output)$ is an array of integers (integer string) with a pre-given length, which was computed during the initiation phase and depends on the machine type and the number of states, or on the number of transitions. The values of this array are in the range $[0 \ldots m-1]$, where m is the size of the output alphabet. Thus, for generating a random string in this form, a random integer in this range must be generated for each array value (Algorithm 21).

3.2.6. Generating an initial set of points

For our search algorithm, the initial set of randomly generated points in the search space is required. In Subsection 3.2.5 the algorithms for generating one random FST were defined.

Algorithm 21 $SR(FST.output)$ random generator in a universe

```
1: $SR[] \leftarrow$new($SR[Multiverse.SRoutputLength]$)
2: **if** $m > 0$ **then**
3:     **for** $i = 0 \rightarrow sr[].size - 1$ **do**
4:         $SR[i] \leftarrow randomInteger \in [0 \ldots m - 1]$
5:     **end for**
6: **else**
7:     $SR[] \leftarrow [0]$
8: **end if**
9: **return** $SR[]$
```

The number of randomly generated points, i.e. the size of initial set is characterized by two parameters – $pc1$ and $pc2$:

- $pc1$ shows the percentage of $cSR(FST.transition)$ generated out of the $Universe.size$,
- $pc2$ shows the percentage of $SR(FST.output)$ generated out of all the possible output functions for each $cSR(FST.transition)$.

Algorithm 22 Generator of an initial random set of points

```
**Require:** $pc1$
**Require:** $pc2$
1: $nT \leftarrow \lfloor Universe.size \times pc1 \rfloor$
2: **for** $i = 0 \rightarrow nT - 1$ **do**
3:     $coordT \leftarrow$generateRandomCSR($FST.transition$)
4:     $nO \leftarrow \lfloor NumberOfAllPossibleOutputFunctions \times pc2 \rfloor$
5:     $keyU \leftarrow Universe.iRange$;
6:     $keyT \leftarrow transform(coordT)$;
7:     **for** $i = 0 \rightarrow nO - 1$ **do**
8:         $coordO \leftarrow$generateRandomSR($FST.output$)
9:         $keyO$ =transform($coordO$);
10:        $key \leftarrow [keyU, keyT, keyO]$
11:        **if** POINTS.containsKey($key$) **then**
12:            $point \leftarrow$ point($coordT, coordO$)
13:            $point.score \leftarrow$evaluate($point$)
14:            POINTS.put($key, point$)
15:        **end if**
16:    **end for**
17: **end for**
```

For certain amount of $cSR(FST.transition)$ (defined by $pc1$), a random $cSR(FST.transition)$ is generated and for this transition function in turn, several random output functions $SR(FST.output)$ are generated. The number of such functions is defined by $pc2$. We define point $cSR_S(FST) = \{cSR(FST.transition), SR(FST.output)\}$ and key. If this point in the search space had not yet been generated, we evaluate it and add to the associative array with this key (Algorithm 22).

3.2.7. Visualization

The basic methods used for visualizing the search process are presented below. Due to the search space being multidimensional, techniques that reduce the dimensionality are required, in order to represent our search space in 2D or 3D graphs.

Representing function value by color

The motivation behind the colorizing process is to reduce the dimensionality of the graph by transforming one dimension into color. For example, this allows to reduce from a 3D graph (point coordinate x, point coordinate y, value at point) to a 2D graph (point coordinate x, point coordinate y, color of the point) (Figure 3.9).

This reduction is done by transforming the value of the function, which for our problem is $\in [0 \ldots 1]$ into a corresponding color (Figure 3.10).

This transformation is defined by function $Color.getHSBColor()$, which constructs a point in the color space represented by Hue, Saturation and Brightness (HSB color model). For our purposes, the hue value is fixed to 1.0 (red) and only the saturation and brightness values are changed. This allows us to modify the colors from black to red. We also added a specific color – white – for the points with the value 1.0 to add more contrast.

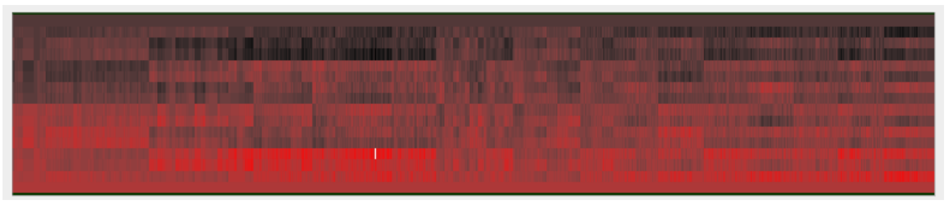


Figure 3.9 Reducing 3D graph into 2D

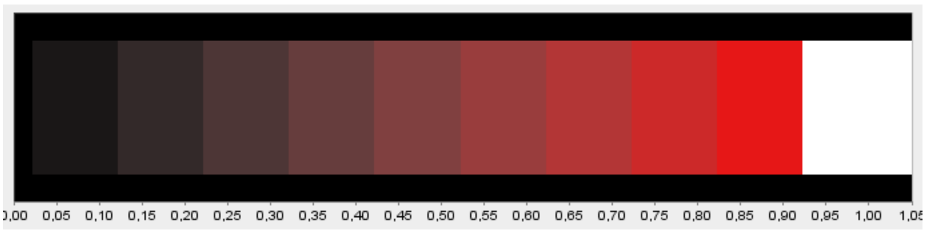


Figure 3.10 Function $\in [0 \ldots 1] \rightarrow Color.getHSBColor(1.0f, value, value)$

Visualizing FSM search space

We can represent each FSM with two vectors of integers ($\mathrm{cSR_S(FST)}$ representation) and for each vector we can find a corresponding number. Thus, each FSM can be represented by two numbers (see Section 2.7). Where $\mathrm{cSR_D(FST)}$ is used, there is only one number, which corresponds to transition function of the FSM. The objective function sets the score value for each machine in the range $[0 \ldots 1]$. We are interested in a diagram, which demonstrates the relation of the FSM and its score.

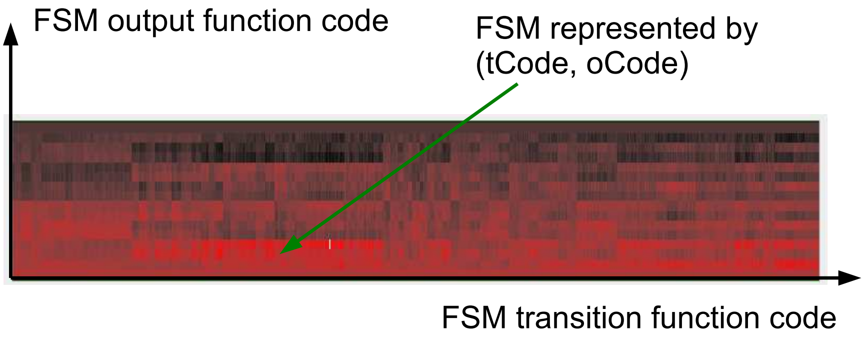


Figure 3.11 2D score graph, where FST is represented by 2D point and its value by color

2D Score graph. For the $\mathrm{cSR_S(FST)}$ coding system, there are 3 coordinates for each point. Two numbers are for FSM representation (i.e. the transition function code and output function code) and one number for its score. In order to draw a two-dimensional diagram, the dimension reduction method described in Section 3.2.7 can be used. Figure 3.11 provides an example of this diagram, where each point corresponds to one FSM, where the number on the horizontal axis stands for the transition function code and the number on the vertical axis corresponds to the output function code. The FSM score is illustrated by color of the point.

1D Score graph. For the $\mathrm{cSR_D(FST)}$ representation, we can omit one dimension, namely the output function code. Only two dimensions remain – the transition function code and the FSM score value.

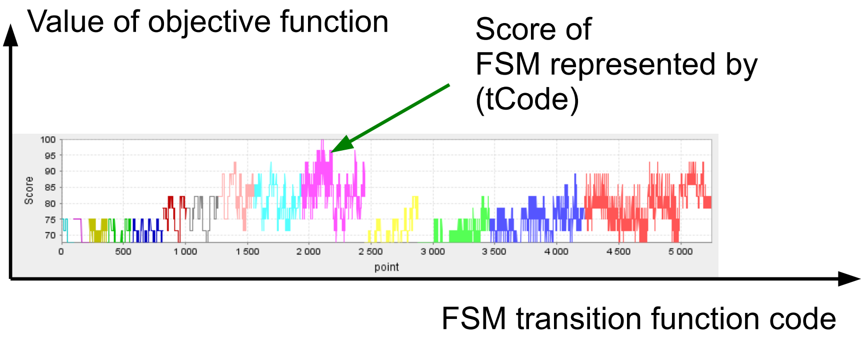


Figure 3.12 1D score graph, where FST *is represented by point and value*

Figure 3.12 demonstrates the visualization example, where the coordinates on the horizontal axis show the transition function code and the coordinates on the vertical axis show the FSM score value. The different colors on the chart demonstrate the different subspaces.

3.3. Search Algorithm

The task of the search algorithm is to find a point in a search space that corresponds to the FST with the optimal behavior. For each point in a search space, the evaluation function assigns a score value from $[0 \ldots 1]$, which describes how well the corresponding FST behaves. Thus, the task is to find the point with the maximal value (1.0) or alternatively, at least the point with the maximal score value.

The search space (multiverse) consists of several universes, i.e. non-intersecting subsets, and each universe contains points (see Section 3.2). Some of the universes have a higher probability of containing the solution than others. The objective is to subdivide the search algorithm into two phases (Algorithm 23).

The first phase chooses the universe which is 'better' than others, while the second phase searches this universe locally in order to find the best point:

1. Phase 1. 'Meta search'. During the meta search, we try to evaluate the universes (subspaces), in order to subsequently enable us to choose the universe, which is more likely to include solutions (Section 3.4).

2. Phase 2. 'Universe local search'. The aim of the local search is to find the maximal point inside a pre-given universe (Section 3.5).

Algorithm 23 Searching in $cSR_S(FST)$ search space

```
initiate(Multiverse)
while (solution not found) and(there are not explored subspaces) do
    $bestSubspace$ ←metaSearch($Space$)
    search($bestSubspace$)
end while
```

The general objective of entire search process is presented in Figure 3.13. To begin with we can assign a value, for each universe, which shows the quality of subspace. After the meta search phase, one subspace that has not been explored and has the best value, is selected. During the local search phase, the chosen universe is explored. If the solution is not found in this universe, the algorithm returns to the meta search phase. This cycle is carried out until the solution with maximal value is found or all the universes are explored. If all subspaces have been explored and the solution with the value 1.0 has not been found, then the algorithm returns the solution with the maximal value (< 1.0).

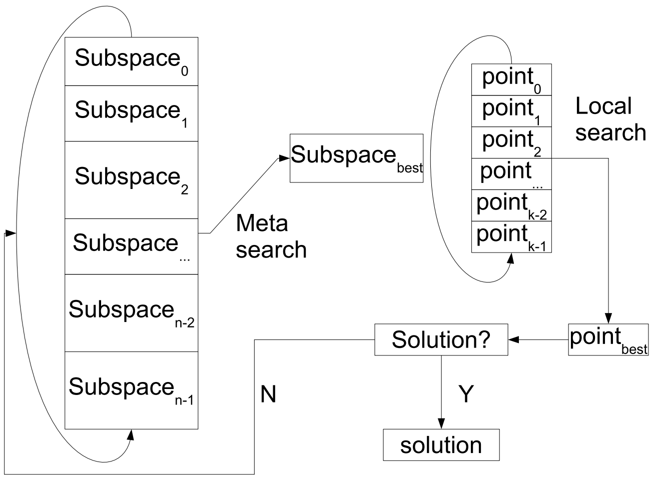


Figure 3.13 General description of the search algorithm

3.4. Multiverse Meta Search

Due to the search space structure, the separate search of the universes can be executed either in parallel or sequentially. Moreover, the ”meta search” algorithm can be employed to define which subspaces to search first. In theory, there is no need to search all the subspaces , but those that are more likely to contain solutions.

The task of meta search algorithm is to assign value to each subspace (described by flag) and to choose 'the best' subspace, which contains the solution with higher probability.

The only useful information known about a subspace is its flag. The flag provides the pattern for the sequences and ultimately defines how the states are connected, which represents certain part of transition function.

An ideal meta search algorithm evaluates the subspaces without generating points in any of them.

However, a realistic meta search algorithm evaluates the subspaces based on the generated points (Algorithm 24).

Algorithm 24 Meta search. Evaluating subspace by values of its points

```
 1: for all $Universe \in Multiverse$ do
 2:     Generate randomly initial set of points $initiate(pc1, pc2)$
 3:     for all $point \in Universe.POINTS$ do
 4:         decode($point \to cSR_S(FST) \to FST$)
 5:         $point.value \leftarrow$getScore($FST$)
 6:     end for
 7:     $Universe.value \leftarrow$ evaluateUniverse($Universe.POINTS$)
 8: end for

 9: function bestUniverse
10:     return the Universe with maximal score, which is not explored
11: end function
```

The simplest idea is to evaluate the universe subspace based on the average value of all of its points. The notion behind this method is to randomly generate a certain amount of points defined by their percentage of the size of the search space. Consequently, the subspace score is constructed on the basis of the values of those points. This method uses the average function for obtaining the score.

$$Universe.score = \frac{\Sigma_{k=0}^{Points.size-1} point[k].value}{Points.size}$$

As with the previous method, we randomly generate a certain amount of points defined by their percentage of the size of the search space and subspace score based on the points values. This method uses the maximal function instead of average value to get the score.

$$Universe.score = Maximum(Points[k].value)$$

Example 3.5 *Suppose the task is to construct a Moore machine which is consistent with the 'aab recognizer' training set (Figure 4.8). After initialization (see Figure 3.4), there are 14 subspaces. In each subspace 10% of points are* *randomly generated (see Algorithm 5). Subsequently, the average value over all in generated points in subspace is computed (see Figure 3.14).*

```
(1, 3, 5): * 1 * 2 * 3 * *  av.m.=0.8333333333333333
(1, 3, 4): * 1 * 2 3 * * *  av.m.=0.8728632478632479
(1, 2, 5): * 1 2 * * 3 * *  av.m.=0.8525641025641028
(1, 2, 4): * 1 2 * 3 * * *  av.m.=0.8859649122807018
(1, 2, 3): * 1 2 3 * * * *  av.m.=0.8669871794871793
(0, 3, 5): 1 * * 2 * 3 * *  av.m.=0.8333333333333336
(0, 3, 4): 1 * * 2 3 * * *  av.m.=0.8584401709401708
(0, 2, 5): 1 * 2 * * 3 * *  av.m.=0.8434065934065933
(0, 2, 4): 1 * 2 * 3 * * *  av.m.=0.8566126855600537
(0, 2, 3): 1 * 2 3 * * * *  av.m.=0.84981684981685
(0, 1, 5): 1 2 * * * 3 * *  av.m.=0.8388278388278388
(0, 1, 4): 1 2 * * 3 * * *  av.m.=0.8582875457875456
(0, 1, 3): 1 2 * 3 * * * *  av.m.=0.8408851422550058
(0, 1, 2): 1 2 3 * * * * *  av.m.=0.8432757718472003
```

Figure 3.14 Scores of all subspaces for 'aab recognizer' task based on average values of 10% of the points

Now each subspace has a corresponding score that characterizes it. Therefore, the 'meta search' algorithm returns the subspace with the maximal value, if this subspace had not yet been explored. In this example, the first such flag is $(1, 2, 4)$ *with the value* 0.88*, if during the local search step solution is not found, the next subspace is* $(1, 3, 4)$ *with the value* 0.87*, etc.*

Figure 3.15 illustrates the method that uses the maximal function instead of the average value in order to get score in the 'Up-down counter' task example.

```
(1, 3, 5): * 1 * 2 * 3 * *  Av. mass: 0.2586 Max mass: 0.4444
(1, 3, 4): * 1 * 2 3 * * *  Av. mass: 0.2407  Max mass: 0.4444
(1, 2, 5): * 1 2 * * 3 * *  Av. mass: 0.2489  Max mass: 0.4603
(1, 2, 4): * 1 2 * 3 * * *  Av. mass: 0.2638  Max mass: 0.5238
(1, 2, 3): * 1 2 3 * * * *  Av. mass: 0.2360  Max mass: 0.4126
(0, 3, 5): 1 * * 2 * 3 * *  Av. mass: 0.2520  Max mass: 0.4761
(0, 3, 4): 1 * * 2 3 * * *  Av. mass: 0.2522  Max mass: 0.5555
(0, 2, 5): 1 * 2 * * 3 * *  Av. mass: 0.2639  Max mass: 0.5714
(0, 2, 4): 1 * 2 * 3 * * *  Av. mass: 0.2591  Max mass: 0.5396
(0, 2, 3): 1 * 2 3 * * * *  Av. mass: 0.2475  Max mass: 0.5238
(0, 1, 5): 1 2 * * * 3 * *  Av. mass: 0.2500  Max mass: 0.6349
(0, 1, 4): 1 2 * * 3 * * *  Av. mass: 0.2448  Max mass: 0.6349
(0, 1, 3): 1 2 * 3 * * * *  Av. mass: 0.2467  Max mass: 0.7142
(0, 1, 2): 1 2 3 * * * * *  Av. mass: 0.2495  Max mass: 0.6507
```

Figure 3.15 Scores of all subspaces for 'Up-down counter' task based on average values of (5%,5%) of the points

3.5. Universe Local Search

The Discrete Gravitational Swarm Optimization algorithm applies the ideas of the Particle Swarm Optimization algorithm (Subsection 3.1.1) and of the Gravitational Search Algorithm (Subsection 3.1.3). Although both these algorithms are intended for continuous search space, we adapt their main ideas to the discrete search space.

Each point in a search space is characterized by:

- $position[]$ – the solution,
- $velocity[]$ – the information about the change of the $position[]$ vector,
- $mass$ – the score value of the solution,
- $position[]_{best}$ – the best known position for the object point,
- $position[]_{global}$ – the best known position in the explored search space.

First of all, the distance between the points in the search space is defined. In our case, each point is the n-dimensional vector of integers. In order to calculate the distance between the vectors, we use the distance in $dimension\ D$ (Algorithm 25), which returns '1', if the values in the corresponding dimension are not equal, otherwise it returns '0'.

Algorithm 25 $distanceInD(valueD_1, valueD_2)$

```
1: if $valueD_1 \neq valueD_2$ then
2:     $distanceD = 1$
3: else
4:     $distanceD = 0$
5: end if
6: return $distanceD$
```

The distance between the vectors is defined as the sum of distances for all dimensions (Algorithm 26).

Algorithm 26 $distance(position_1[], position_2[])$

```
1: for $dimension = 0 \rightarrow position.length - 1$ do
2:     $distance \leftarrow distance +$ distanceInD($position_1[dimension]$, $position_2[dimension]$)
3: end for
4: return $distance$
```

Subsequently, we need to redefine two operations – movement(Algorithm 27), i.e change of position and acceleration (Algorithm 28), i.e. change of velocity.

In the PSO and GSA algorithms, movement and acceleration are defined as sums, However, we use algorithms similar to the 'crossover' operator in the Evolutionary Algorithm.

Algorithm 27 $Move(position[], velocity[], mass_{inertial})$

```
1: for $dimension = 0 \to position.length - 1$ do
2:     if Random $< 1 - mass_{inertial}$ then
3:         $newPosition[dimension] \leftarrow velocity[dimension]$
4:     else
5:         $newPosition[dimension] \leftarrow position[dimension]$
6:     end if
7: end for
8: return $newPosition[]$
```

The 'move' operator (Algorithm 27) changes the current position into a new one according to the tendency, which is described by the $velocity[]$ vector. The ability of changing is described by $mass_{inertial}$. A bigger $mass_{inertial}$ means that this point tends to save its current position. The 'move' operator is similar to the uniform crossover operator between the $position[]$ and $velocity[]$ vectors.

Algorithm 28 $Accelerate$

```
for $dimension = 0 \to velocity.length - 1$ do
    if $Random() < Force_p$ then
        $newVelocity[dimension] \leftarrow position_{pBest}[dimension]$
    else
        $newVelocity[dimension] \leftarrow velocity[dimension]$
    end if
end for
for $dimension = 0 \to velocity.length - 1$ do
    if $Random() < Force_g$ then
        $newVelocity[dimension] \leftarrow position_{gBest}[dimension]$
    else
        $newVelocity[dimension] \leftarrow velocity[dimension]$
    end if
end for
return $newVelocity[]$
```

More details on the *Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions* can be found in [58]. It is similar to the method we use for local search and which also adapts the ideas of combining the PSO and GSA. The *Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions* algorithm was benchmarked against stochastic hill climbing and standard PSO on the *Diophantine Equation Solver* problem.

3.6. Conclusion

This chapter covers the heuristic search algorithm in the context of FSM induction. We presented methods for search space initialization and visualization, operators of the new search algorithm and the algorithm itself.

Canonical string representation of FSM allows to divide the search space into non-intersecting parts. In addition, such representation creates one-to-one correspondence between FSM and a triple of numbers, which can be used as 'key' for the hash function. Hashing of point helps to reduce the number of FSM evaluations.

The specifics of the search space structure provides the opportunity to create the search algorithm that has two phases – the first one for selecting the subspace with a higher probability of containing the solution, and the second phase being the local search inside the selected subspace. The local search algorithm is based on the new 'Discrete Gravitational Swarm Optimization algorithm', which was constructed as a hybrid of GSA and PSO and adapted for the discrete search space.

4. APPLICATIONS

This chapter describes the set of tasks that can be used for benchmarking a proposed method for FSM identification, e.g. *System identification* (see Section 4.1), *Artificial ant problem* (see Section 4.2) and *Binary sequence predictor* (see Section 4.3).

4.1. System Identification

In this section, the application of FSM inference to the system identification problem is discussed. A brief overview of the system identification problem is given, along with defining different objective functions and presenting several experimental results.

4.1.1. Description

The system identification problem is defined as constructing a model of the system, i.e. internal representation, which simulates its external behavior (Figure 4.1). The only data that can be used for model inference are the observable inputs and outputs of the system.

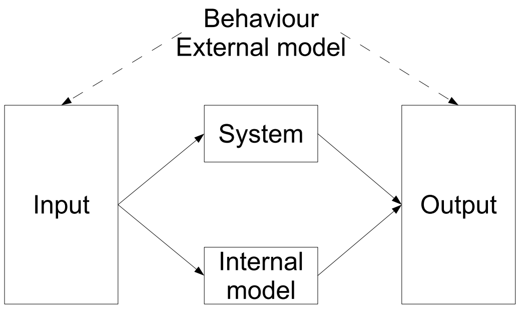


Figure 4.1 System identification

In this research the focus is only on models which can be considered as a finite state machine with an output function. The methods for inferring any structures for language recognition, such as automata or grammars are part of *grammatical inference* and thus not discussed in this context.

The inference from the example (I/O pairs) to general (Model) can be considered as an *inductive inference*. In addition, as the learner cannot control the data it receives, this is a *passive inference*.

Although, only FST is discussed, the two issues relating to classification and identification are someway similar – the inference of a FA can be considered as part of the FST inference).

There are two main FSM properties that are important:

- **The** FSM **size:** if the size of the inferred FSM, i.e. its number of states is not limited, then the problem becomes trivial. There are two problem definitions:

  – Finding a minimum size deterministic FSM that is consistent with the set of given samples.

  – Finding a deterministic FSM with n or less states that is consistent with the set of given samples.

  The problem of finding minimal FSM is more complex.

- **Generalization.** The term 'generalization' denotes the ability to identify unseen data, which is not presented in the training set. Here we can formulate two different goals:

  – to find a **generalized solution**, which performs correctly for all possible I/O sequences, or

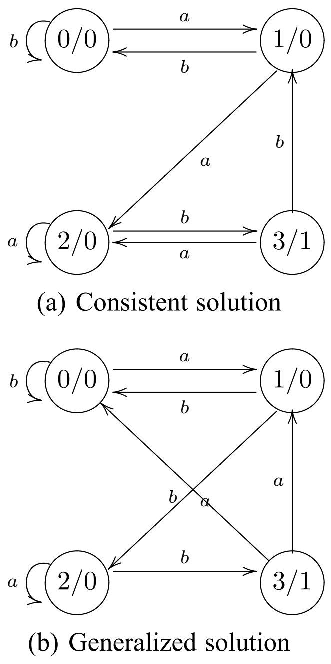
  – to find a **consistent solution**, which performs correctly only for I/O sequences used in inference process (the training set).

  Example 4.1 shows the difference between these two problem statements.

Example 4.1 *Figure 4.2 illustrates two solutions that are consistent with the given training set. The difference lies in the behavior of these models:*

- *the generalized solution (Figure 4.2(a)) behaves as the correct 'aab' recognizer for all possible I/O pairs,*

- *the consistent solution (Figure 4.2(b)) acts as the correct one only for training data, but makes errors for other I/O pairs. For example for the input "aabab" it will output "000101", which is incorrect.*



(a) Consistent solution

(b) Generalized solution

Figure 4.2 Pattern recognizer for 'aab'. Generalized and consistent solution

The problem statement

We formulate our problem as having to find a deterministic FST with n states, which is consistent with the given training set, where:

- The *environment* is defined as the set of I/O pairs. The input and output alphabets can be constructed by parsing the training data.
- The *acting agent* is defined as a FST. Here we consider on both Mealy and Moore machine. The number of states is predefined by the user.
- The search algorithm employs the representations we use, namely $\mathrm{cSR_D(FST)}$, where the output function can be reconstructed from observable output, and $\mathrm{cSR_S(FST)}$.
- The *score* assigned for each FST shows how accurately this machine describes the training data. The *objective function* for this problem is defined in Subsection 4.1.1.

Complexity

The problem of finding a minimum size deterministic FSM that is consistent with the given set of given is equivalent to the problem of determining whether there exists a k-state DFA that is consistent with a set of labeled strings. The identification of minimal consistent automata from the given data is NP-Complete [2]. The problem can be solved in polynomial time on the input size if all strings with a length n or less are given, but remains NP-complete if a small fixed fraction of these strings is missing. In addition, assuming that $P \neq NP$, it is shown that for any constant k, no polynomial time algorithm can be guaranteed for finding a consistent DFA with fewer than opt^k states, where opt is the number of states in the minimum state DFA consistent with the sample [59].

More information about the complexity of FA induction can be found in [60], [61].

Objective function

The main notion behind of constructing the objective of a function for the problem of FST inference from the training set lies in measuring the difference between the expected output of the model known from the output part of the behavior and the output generated by model which is FSM generated by the learner (Figure 4.3).

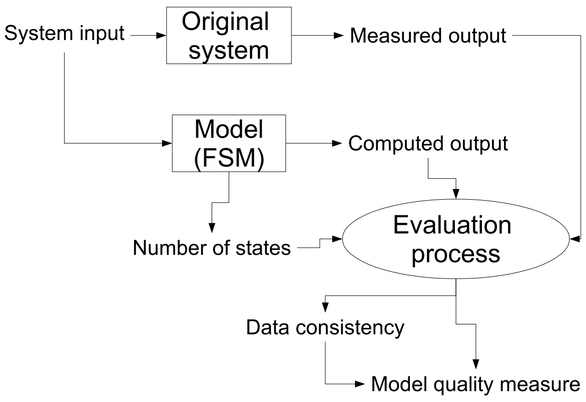


Figure 4.3 FSM *quality evaluation process*

While the system behavior is described by the set of I/O strings, the *objective function* (see Figure 4.4) is constructed based on measuring string distances.

We defined two functions – *Hamming similarity* and *Length of maximal equal prefix* (see Subsection 2.1.1). Based on these functions we can construct *objective functions* by applying one of the string distance functions for each I/O pair and summing them up.

Each *objective function* (Equation 4.2 and 4.1) returns a value $\in [0..1]$.

Input	Expected output	Produced output	Distance
In_0	$Out_0^{expected}$	$Out_0^{produced}$	$distance_0$
In_1	$Out_1^{expected}$	$Out_1^{produced}$	$distance_1$
$In_{...}$	$Out_{...}^{expected}$	$Out_{...}^{produced}$	$distance_{...}$
In_n	$Out_n^{expected}$	$Out_n^{produced}$	$distance_n$

Figure 4.4 Measuring objective value

Objective function $ObjValue_{Ham}$ based on the *Hamming similarity* (Definition 2.12) can be computed by:

$$ObjValue_{Ham} = \frac{\Sigma_{i=1}^{n}(S_{Ham}(Out_i^{expected}, Out_i^{produced}))}{\Sigma_{i=1}^{n} Length(Out_i^{expected})}. \tag{4.1}$$

Objective function $ObjValue_{LP}$ based on *length of maximal equal prefix* (Definition 2.13) is defined by

$$ObjValue_{LP} = \frac{\Sigma_{i=1}^{n}(D_{LP}(Out_i^{expected}, Out_i^{produced})}{\Sigma_{i=1}^{n} Length(Out_i^{expected})}. \tag{4.2}$$

4.1.2. Examples

This section includes experiments with and examples of applying the FST search algorithm for solving the system identification problem.

’ab’ Recognizer. Induction by enumeration

The task of the *’ab’ Recognizer* is to reconstruct the Moore machine M_{abRec} ’ab recognizer’ from the training set (Figure 4.5). The M_{abRec} machine must output ’1’ in the output sequence, if the pattern ’ab’ was found in the input string. Otherwise, the output is ’0’. The FST has 3 states, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$.

```
abbbbb, 0010000
aababa, 0001010
babbba, 0001000
abbbbb, 0010000
```

Figure 4.5 Training set for the ’ab’ recognizer task

For such small search spaces with 216 possible transition functions and 8 possible output functions, it is cheaper to sequentially check all possible machines and then choose the best one.

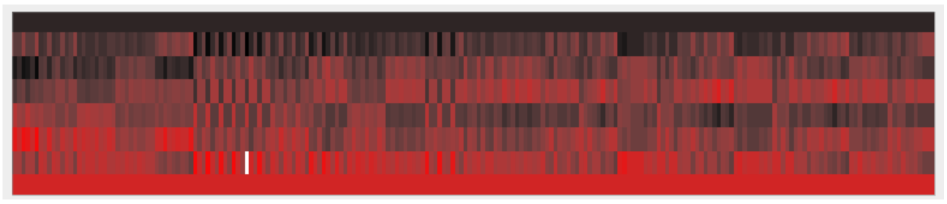


Figure 4.6 Search space landscape for $cSR_S(M_{abRec})$ representation

Figure 4.6 shows the score graph for all possible FST described by $\mathrm{cSR_S(FST)}$ representation and objective function $ObjValue_{Ham}$. The white point belongs to the optimal solution represented by $cSR_S(M_{abRec}) = [1, 0, 1, 2, 1, 0], [0, 0, 1]$ with the score 1.0. Here the algorithm enumerates all 216×8 points.

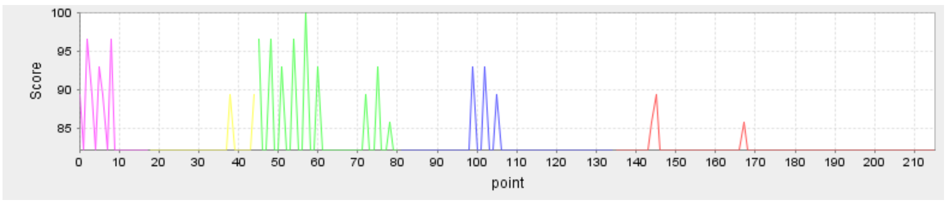


Figure 4.7 Search space landscape for $cSR_d(M_{abRec})$ representation

Figure 4.7 demonstrates the score graph for the situation, where the FST is represented by $\mathrm{cSR_D(FST)}$ representation. Thus, the output function of the machine is inferred from I/O pairs, and the point with 100% value (objective value 1.0) belongs to the solution coded by $cSR_d(M_{abRec}) = [1, 0, 1, 2, 1, 0]$. The algorithm only checks 216 points.

Although such representation is applicable only in situations, where output function can be recomputed, it is simpler to apply the representation $\mathrm{cSR_D(FST)}$ in order to reduce the size of the search space.

’aab’ Recognizer

The *’aab’ Recognizer* task is to construct the Moore machine M_{aabRec} from the training set (Figure 4.8), which will behave as ’aab Recognizer’. The M_{aabRec} machine must output ’1’ in the output sequence, if pattern ’aab’ was found in the input string. Otherwise the output is’0’. The FST has 4 states, $\Sigma = \{b, a\}$, $\Delta = \{0, 1\}$.

Figure 4.9 shows the set of discovered solutions for the ’aab’ recognizer task, which are presented by $cSR_d(M_{aabRec})$ and were found by enumeration.

Figure 4.10 displays the score graph for all possible machines (5248×16) for $cSR_S(M_{aabRec})$ representation. The solutions are located in the areas marked by a white line.

```
babaabaabaab, 0000001001001
bbaabaaaaaba, 0000010000010
bbbaabbbaabb, 0000001000010
bbaabbaabaaa, 0000010001000
aabbaaaabbbb, 0001000001000
aababaaababb, 0001000001000
```

Figure 4.8 Training set for the 'aab' recognizer task

```
[0, 1, 0, 2, 3, 2, 0, 1]
[0, 1, 0, 2, 3, 2, 1, 1]
[0, 1, 1, 2, 3, 2, 1, 1]
[0, 1, 1, 2, 3, 2, 0, 1]
[1, 2, 1, 0, 3, 2, 0, 0]
[1, 2, 1, 0, 3, 2, 1, 0]
[1, 2, 0, 0, 3, 2, 0, 0]
[1, 2, 0, 0, 3, 2, 1, 0]
```

Figure 4.9 Discovered solutions for the 'aab' recognizer task training set

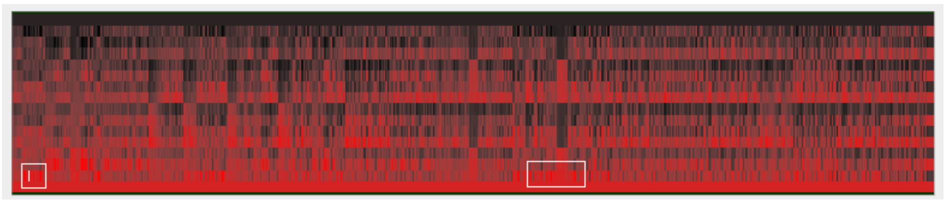


Figure 4.10 Search space landscape for $cSR_S(M_{aabRec})$ representation

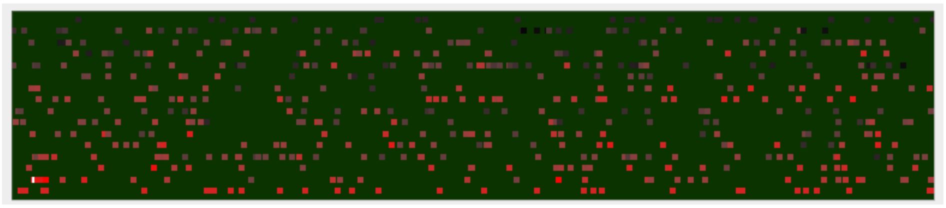


Figure 4.11 Explored points in the search space for $cSR_S(M_{aabRec})$ representation

Figure 4.11 shows the points explored during the search process. (0.1, 0.1) points were randomly generated at the initial stage.

The meta search phase found a universe $\{(1,\ 3,\ 4):\ \ast\ 1\ \ast\ 2\ 3\ \ast\ \ \ast\ \ast\}$ with average mass=0.6068. The search algorithm checked 26 points. The total search covered over 530 points out of 83968.

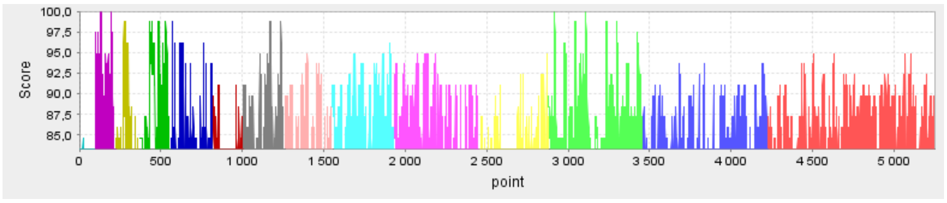


Figure 4.12 Search space landscape for $cSR_d(M_{aabRec})$ representation

Figure 4.12 shows the score graph for all 5248 machines presented by the $cSR_d(M_{aabRec})$. The solutions are concentrated only in two subspaces, defined by flags $\{(1,3,4):\ \ast\ 1\ \ast\ 2\ 3\ \ast\ \ \ast\ \ast\}$ (purple region) and $\{(0,1,4):\ 1\ 2\ \ast\ \ \ast\ 3\ \ast\ \ \ast\ \ast\}$ (light green region).

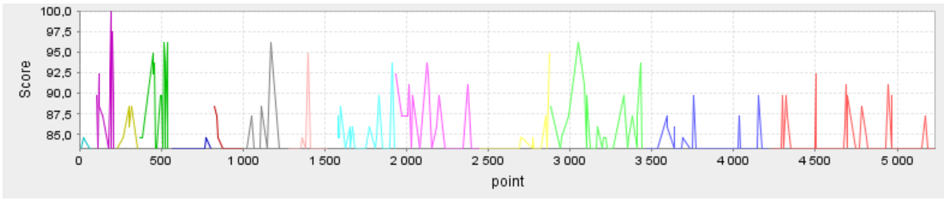


Figure 4.13 Explored points in the search space for $cSR_d(M_{aabRec})$ representation

Figure 4.13 depicts the points explored by the search process. (0.05, 0.05) random FST were generated at the initial phase. The meta search phase found a universe $\{(1,\ 3,\ 4):\ \ast\ 1\ \ast\ 2\ 3\ \ast\ \ \ast\ \ast\}$ with an average mass=0.876 at the second step. The total number of points explored was 266 out of 5248.

Division by two

The *Division by two* task is to construct the Moore machine M_{div2}, which divides numbers in binary form by 2, usually done by bit right shifting, from the training set (Figure 4.14). The FST has 4 states, $\Sigma = \{1,0\}$, $\Delta = \{0,1\}$.

```
  101, 0010
11111, 001111
01001, 000100
  100, 0010
  110, 0011
    1, 00
    0, 00
```

Figure 4.14 Training set for the 'Division by two' task

Figure 4.15 shows the score graph for all possible FST (5248×16). The white point belongs to the solution $cSR_S(M_{div2}) = \{[1, 0, 2, 3, 2, 3, 1, 0], [0, 0, 1, 1]\}$ with the score 1.0.

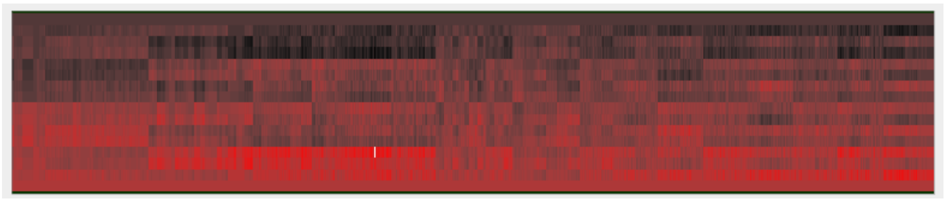


Figure 4.15 Search space landscape for $cSR_S(M_{div2})$ representation

Figure 4.16 demonstrates the set of points explored during the search. The initial phase generated (0.05, 0.05) of random points, and the meta search found a universe $\{(0,\ 2,\ 3) :\ 1\ *\ 2\ 3\ *\ \ *\ \ *\ *\}$ with an average mass=0.5857 (90 points in this Universe were checked). The total amount of explored points was 320 out of 83968.

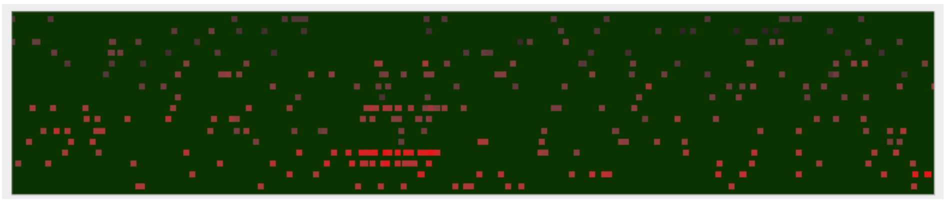


Figure 4.16 Explored points in the search space for $cSR_S(M_{div2})$ representation

Figure 4.17 demonstrates the score graph for all possible points for the $\mathrm{cSR_D(FST)}$ form.

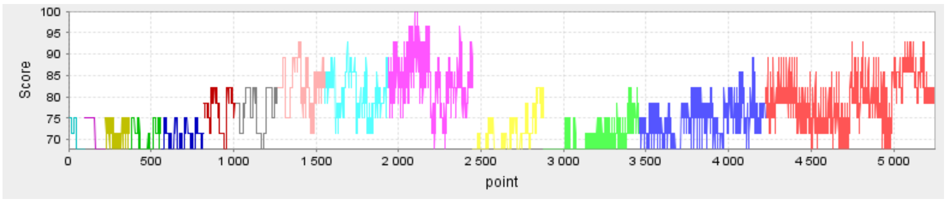


Figure 4.17 Search space landscape for $cSR_d(M_{div2})$ representation

Figure 4.18 shows the set of explored points. During the initiation phase (0.05, 0.05) of points were randomly generated, and the meta search phase found a universe $\{(0,\ 2,\ 3):\ 1\ *\ 2\,3\ *\ \ *\ \ *\ *\}$ with an average mass: 0.835 at the second step. The search algorithm explored 297 out of 5248 points.

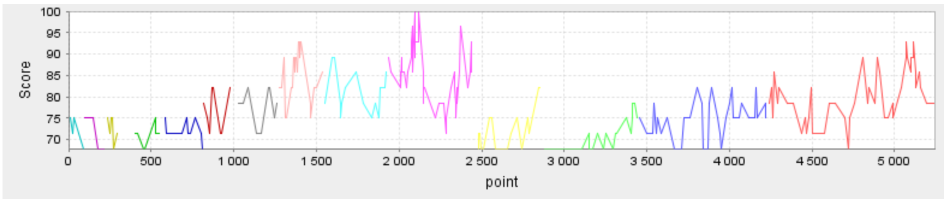


Figure 4.18 Explored points in the search space for $cSR_d(M_{div2})$ representation

Up down counter

The *Up down counter* task is to construct the Moore machine M_{udc} from the training set (Figure 4.19), which analyzes the input sequence in a binary alphabet. Let $w = s_1 s_2 \ldots s_t$ be an input string, $N_0(w) =$ number of 0 in w and $N_1(w) =$ number of 1 in w. Then we have the length of the word $|w| = N_0(w) + N_1(w) = t$. The output of the machine should equal: $r(t) = [N_1(w) - N_0(w)]\ mod\ 4$.

Sample input/output session:

stimulus	11011100
response	012123032

The FST has 4 states, input alphabet $\Sigma = \{0, 1\}$ and output alphabet $\Delta = \{0, 3, 1, 2\}$.

Figure 4.20 demonstrates the score graph for the set of points explored during the search process for the $\mathrm{cSR_D(FST)}$ form (5248 machines). The solution is $cSR_d(M_{udc}) = \{[1,\ 2,\ 3,\ 0,\ 0,\ 3,\ 2,\ 1]\}$.

```
01010101, 030303030
10101010, 010101010
00110011, 032303230
00111100, 032301210
01001100, 030323032
10011001, 010301030
11001001, 012101030
```

Figure 4.19 Training set 'Up down counter'

For the $\mathrm{cSR_S(FST)}$ form the search space is too big for visualization by enumeration (5248×256 machines).

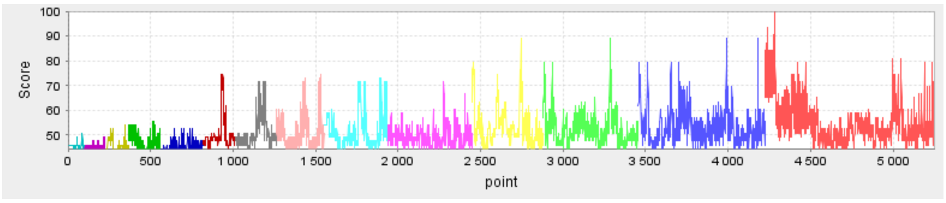


Figure 4.20 Search space landscape for $cSR_d(M_{udc})$ representation

Figure 4.21 shows the points explored during the search. At the initial phase (0.05, 0.05) of the points were generated randomly, and the meta search phase found a universe $\{(0,\ 1,\ 2):\ 1\,2\,3\ *\ *\ *\ *\ *\}$ with an average mass= 0.5353 at the forth step. The algorithm explored 393 out of 5248 points.

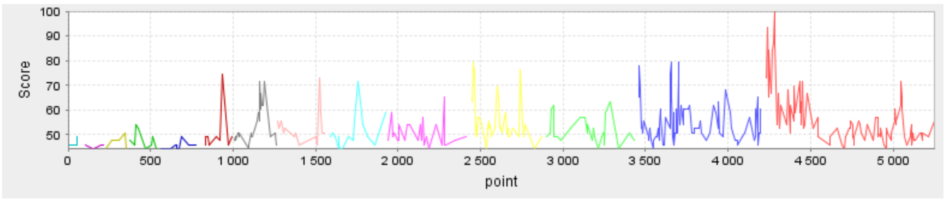


Figure 4.21 Set of points explored in the search space landscape for $cSR_d(M_{udc})$ representation

Figure 4.22 shows the set of points explored during the search process. At the initial phase (0.05, 0.05) of points were randomly generated, and the meta search phase found the a universe $\{(0,\ 1,\ 2):\ 1\ 2\ 3\ *\ *\ *\ *\ *\}$ with an average mass=0.255 at the forth step. hence, in total, the algorithm explored 4780 out of 1343488 points.

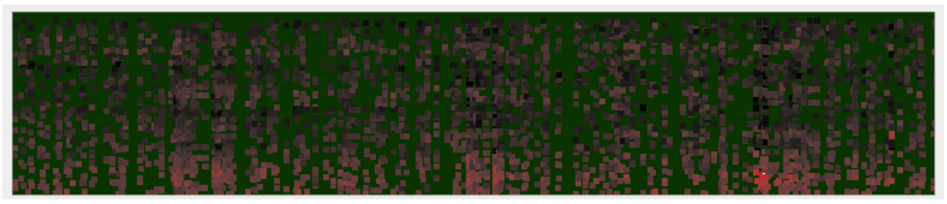


Figure 4.22 Set of points explored in the search space landscape for $cSR_S(M_{udc})$ representation

4.1.3. Package: System Identification

Package `System identification` (Figure 4.23) contains the implementation of the system identification problem.

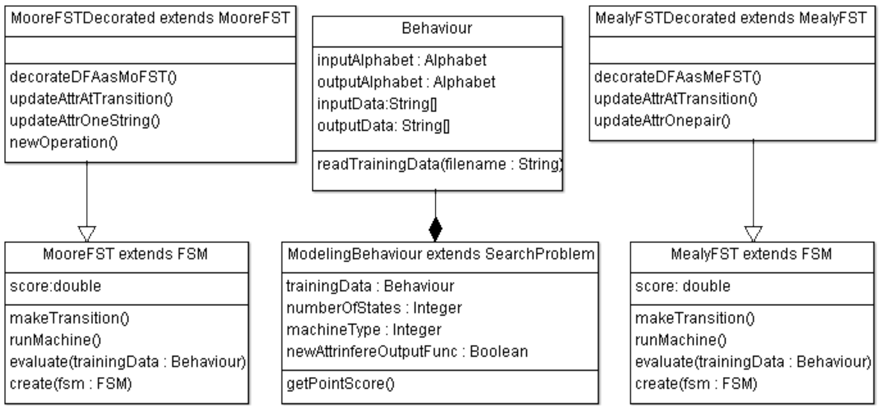


Figure 4.23 Class diagram describing the package 'System identification'

The package contains the following classes:

- `ModellingBehaviour` is the main class that extends the `SearchProblem` class and implements the main methods for FSM evaluation of training data.
- `MooreFST` class extends the `FSM` class and implements the method of the classical Moore machine work-flow (Algorithm 2).
- `MooreFSTDecorated` class extends the `MooreFST` class and implements the algorithm for decorating $\mathrm{DFA}_\emptyset$ as a Moore machine.
- `MealyFST` class extends the `FSM` class and implements the method of the classical Mealy machine work-flow.

- `Behavior` implements the algorithms for reading, parsing and storing information about training data.

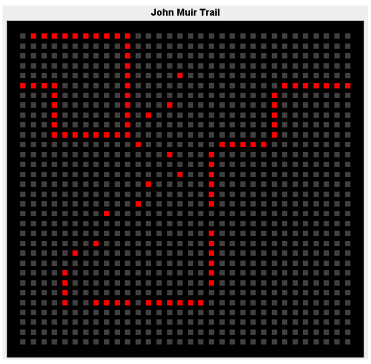
4.2. Artificial Ant Problem

This section examines the *Artificial ant problem* also known as the *Trail tracker problem*. This task is often used for benchmarking Evolutionary Algorithms, not only in a case of FSM inference, but also in the case of Genetic Programming [22] and artificial neural network learning [21].

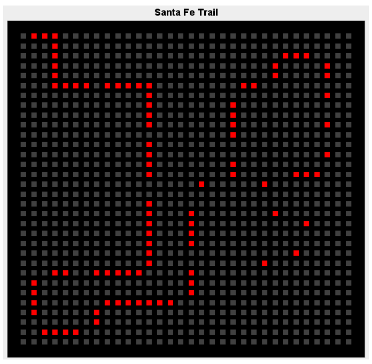
4.2.1. Description

The *Artificial ant problem* was originally proposed by Jefferson et al. [21]. The goal is to construct an agent, which takes information about a subsequent cell, i.e. whether there is food or not, and moves along the grid, with aim of finding the optimal trail for collecting all the food in the grid.

The grid itself is a 32×32 toroidal structure. Therefore, if the ant is at the bottom of such a grid and the next move is 'move down', the following cell will be at the top. The food on the grid is located in a special way, constructed to make the task more complex. The are two well-known trails – the *John Muir Trail* (Figure 4.24(a)) developed by Jefferson et al. [21] and the *Santa Fe Trail* (Figure 4.24(b)). Both of them contain 89 cells of food.



(a) *John Muir Trail*



(b) *Santa Fe Trail*

Figure 4.24 Artificial Ant trails

The ant is modeled by a Mealy-type machine with n states. There is only one input variable with two values (events), because the ant can only see a single cell directly in front of it. Input alphabet contains only two characters:

- 'F' (food) – there is food in the next cell,
- 'E' (empty) – the next cell is empty.

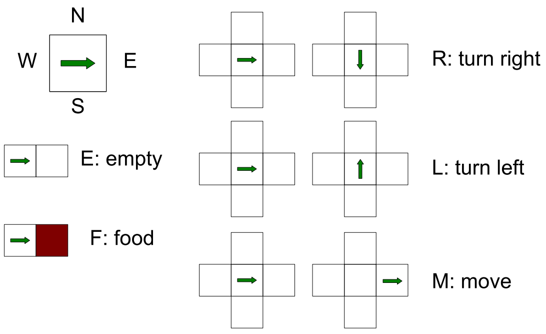


Figure 4.25 Events, actions and orientation

The actions allowed for the ant can be coded by a 4-letter alphabet:

- ’W’ (wait) – the ant will do nothing at this step,
- ’M’ (move) – the ant will move to the next cell and if there is food it will eat it (the eaten food is removed from the grid),
- ’R’ (turn right) – the ant will stay in the same cell, but turn right,
- ’L’ (turn left) – the ant will stay in the same cell, but turn left.

The ant can only see the cell in the front of it, ant the address of the cell is defined by the orientation of the ant:

- ’N’ (north) – the ant can see the top cell,
- ’E’ (east) – the ant can see the right cell,
- ’S’ (south) – the ant can see the bottom cell,
- ’W’ (west) – the ant can see the left cell.

The starting position of the ant on the grid is the top left cell $(0, 0)$. The initial orientation is ’East’. Thus, the input alphabet is $\Sigma = \{E,\ F\}$ and the output alphabet is $\Delta = \{W,\ M,\ L,\ R\}$.

Figure 4.25 demonstrates the defined alphabets, i.e. actions and events, for the situation, where the ant is oriented ’East’.

Objective function

The score of the ant can be found only by running the simulation. The original objective function shows how much food was eaten during 200 steps. We modify this function so that it returns a value $\in [0 \dots 1]$ (Equation 4.3). Therefore, we divide the number of eaten food cells by the total number of food cells on grid.

$$ObjValue = \frac{eaten\ food}{total\ amount\ of\ food\ on\ trail} \tag{4.3}$$

Complexity

The question is how complex must the FSM, i.e. number of states be,in order to be capable of fulfilling the task.

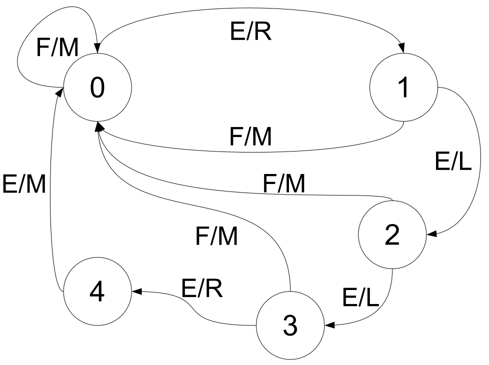


Figure 4.26 Original ant for John Muir Trail

Jefferson et al. [21] constructed the FSM with 5 states (Figure 4.26), which is able to eat 81 pieces of food out of 89 on the *John Muir Trail* by 200 steps and eat all of the food by 314 steps. This ant can be presented by $cSR_S(M_{ant81t314}) = \{[1, 0, 2, 0, 3, 0, 4, 0, 0, 0], [3, 1, 2, 1, 2, 1, 3, 1, 1, 1]\}$.

As is evident, this ant was not able to eat all the food by 200 steps. In the literature ant with up to 13 states are used.

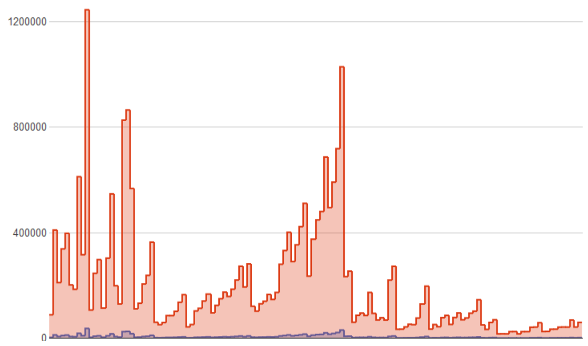
Comparing the effectiveness of solutions

Due to the various methods and algorithms for *artificial ant* representation (grammars [62]), trees, FSMs, neural nets, etc.), the different heuristic search methods, including *Genetic Programming*, *Genetic Algorithm*, *Ant Colony Optimization* [63], and the different possible *evaluation function*s the proposed method cannot be directly compared to the already existing solutions. In addition, we minimized and structured the search space, so it is difficult to separate the effect of the proposed gravitationally-inspired search algorithm from the effect of search space minimization.

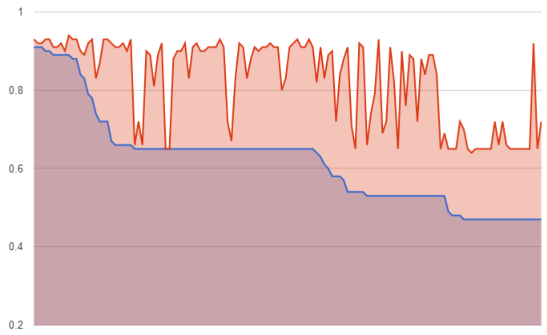
One of the most time-consuming processes in the search algorithm is evaluating the ant. In order to optimize the search process we need to minimize the number of evaluations. *The number of fitness evaluations* (N_{eval}) shows how many times the evaluation function was computed. $\mathrm{N}_{\mathrm{eval}}$ is also applicable to different search methods, so it can be used as a metric for algorithm comparison.

4.2.2. Simulation results. John Muir Trail

During this experiment we will try to construct a Mealy machine with 6 states, which will model our artificial ant. The ant will be simulated on the *John Muir Trail*. The input and output alphabets were defined above, and the objective function counts the number of food cells eaten with 200 steps. The $\mathrm{cSR_S(MeFST)}$ representation will be used.



(a) Number of point generated during the search process



(b) Max value of evaluation function at the initial and final iteration

Figure 4.27 Simulation on John Muir Trail. Artificial ant with 6 states

For an ant with 6 states, the search space contains 3152263549140 points and they are divided into 132 partitions.

Figure 4.27 covers the simulation results:

- Figure 4.27(a) illustrates the number of points generated during search process. The blue line depicts the number of points generated during the initial phase that were used analyzing the partitions, while the red line demonstrates the number of points, that were explored in each universe during the search.

- Figure 4.27(b) illustrates the objective function values at the initial stage, i.e. after generation of random points,as well as at the final stage, namely after the search process for each universe (partition). The partitions are ordered by the multiverse meta search algorithm with respect to the maximal value of the objective function at the initial stage.

One possible solution is:

$$\begin{aligned} cSR_S(M_{JMT}) = \{&[1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 2,\ 0,\ 3,\ 0,\ 0,\ 4], \\ &[1,\ 0,\ 2,\ 0,\ 2,\ 0,\ 0,\ 2,\ 0,\ 0,\ 1,\ 0]\}, \end{aligned}$$

with the objective function value 0.944, signifies the ant was able to eat 84 pieces of food out of 89 by 200 steps, while all of the food can be eaten by 236 steps.

4.2.3. Simulation results. Santa Fe Trail

During the experiments we constructed MeFSTs with $5 \ldots 7$ states, which model the *artificial ant*. The ant is simulated on the *Santa Fe Trail*. The input and output alphabets are defined as $\Sigma = \{E,\ F\}$ and $\Delta = \{M,\ L,\ R\}$, the objective function counts the number of food cells eaten during 400 steps using Equation (4.3).

The parameters for the proposed DGSO method are the following:

- the number of steps in each local search phase $e = 50$,
- at the initial stage of the *multiverse* search , a certain number of points for the *universe* must be generated. Coefficients C_t (coefficient for the transition function) and C_o (coefficient for the output function) (Table 4.1) show how many points are generated:
  - $|Universe(MeFST.transition)| = |cSR_s(MeFST.transition)| \cdot C_t$
  - $|Universe(MeFST.output)| = |cSR_s(MeFST.output)| \cdot C_o$

Table 4.1 Initialization parameters

n	C_t	C_o
5	0.002	0.002
6	0.00015	0.00015
7	0.00005	0.00005

We use $\mathrm{N}_{\mathrm{eval}}$ (see Subsection 4.2.1) for measuring the quality of the proposed DGSO method. Christensen, S. and Oppacher, F. [64] proposed a method with $\mathrm{N}_{\mathrm{eval}} = 20696$ fitness evaluations for the *Santa Fe Trail* based on Genetic Programming + small tree analysis. In Table 4.2 the third column contains the results of the method proposed by Chivilikhin et al. [63] with respect to the number of states in the FSM.

Table 4.2 Fitness evaluations

Christensen ([64])	n	*Chivilikhin ([63])*	DGSO (avg)	DGSO (min)
20696	5	*10975*	114912	**4642**
20696	6	*9313*	233508	**5205**
20696	7	*9221*	423208	93290

Table 4.2 contains the results of a proposed DGSO method, showing the mean number of evaluations and the minimal number of evaluations in 100 runs. The arithmetic mean of $\mathrm{N}_{\mathrm{eval}}$ for the DGSO method is bigger than for the Chivilikhin [63] and Christensen [64] methods, but the minimal $\mathrm{N}_{\mathrm{eval}}$ is better for the case of 5–6 states.

4.2.4. Analysis

Despite the fact that the mean number of evaluations $\mathrm{N}_{\mathrm{eval}}$ is bigger than for the other existing methods, the proposed DGSO method has significant potential and in some cases, was able to find the solution with a smaller $\mathrm{N}_{\mathrm{eval}}$. The results with a smaller $\mathrm{N}_{\mathrm{eval}}$ were produced when the method found the correct *universe* containing the solution during meta-search stage at the first step. The results with a bigger $\mathrm{N}_{\mathrm{eval}}$ were produced during runs, where several *universes* (25 for the case of 5 states and 34 for the case of 6 states) were searched before the solution was found.

There are two main problems with the proposed method that lead to such a big $\mathrm{N}_{\mathrm{eval}}$:

- During the initialization process, a certain amount of points in each *universe* must be generated (for test cases these parameters are presented in Table 4.1). At present, this amount is defined with respect to the size of the *universe*. If the search space is big, as is the case for $n = 7$, a large number of points is already generated and evaluated at the initial stage.

- The worst-case scenario for the DGSO search method is the situation where all the *universe*s are searched and the last one contains the solution, provided there even is one. The best-case scenario is the situation where the solution is found in the first *universe*. Currently, the *universe*s are evaluated based on the best point found during the initialization phase, which is not optimal.

A possible solution for these two problems is to change the initialization process and modify the function for *universe* evaluation. The ideal situation would be the possibility for evaluating the *universe* without having to generate points. Such optimization can be researched in the future.

4.2.5. Package: Trail tracker

This package contains the implementation of the *Trail tracker problem*:

- `Grid` class stores the grid values.
- `Trail` class implements the functionality required for trail processing, i.e. reading trail, and uses `Grid` class for storing trail information.
- `TrailTracker` class extends the `FSM` class, which implements the functionality required for moving the ant on the trail, and uses `Grid` for updating the path.
- `TrackTheTrail` is the main class of the package (extends the `SearchProblem`), which creates the trail from the data file and runs the artificial ant on the grid.

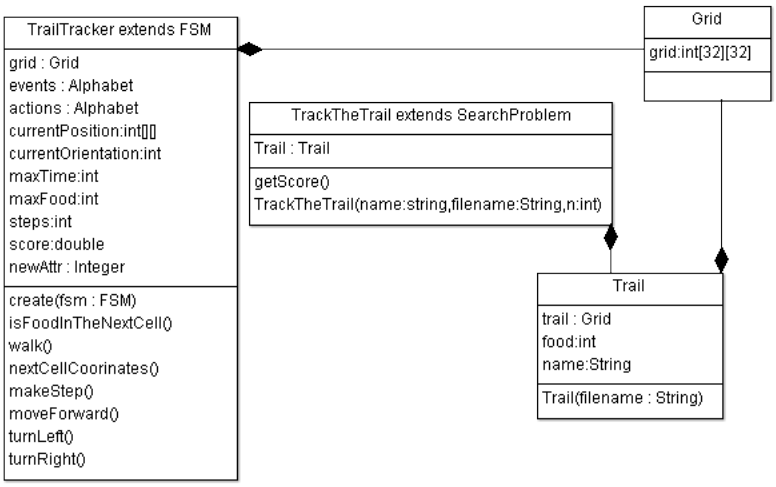


Figure 4.28 Class diagram describing the 'Trail tracker' package

4.3. Binary Sequence Predictor

This section discusses one of the benchmarks, initially used for benchmarking the *Genetic Programming* algorithm.

4.3.1. Description

The world can be considered as an environment and one of the main abilities of living beings is the capacity to adapt their environment, an ability essential for survival. In the task at hand, we will try to model this situation.

This game was initially introduced by Fogel in 1966 [40] to demonstrate the evolution of FST thanks to *Genetic Programming*.

In more detail, the simplest living being is modeled by a Mealy machine and the environment is a bit string with a certain periodical bit mask. The value of one bit is passed on to the Mealy machine at particular moment of time. The task is to predict the next value in the sequence. In addition, this task can be seen as the prediction of binary sequences. There are several heuristic search algorithms, that can be used in this context, e.g. generated simulated annealing [28] or *Evolutionary Algorithms* [27].

Representation

The agent here is modeled as a Mealy machine with the input alphabet $\Sigma = \{0,\ 1\}$ and the output alphabet $\Delta = \{0,\ 1\}$, therefore $\mathrm{cSR_S(MeFST)}$ is used here.

Complexity

In the trivial case we can choose a number of states equal to the size of the bit mask, used to generate bit string. Although this gives a 100% precise prediction, the objective is to find a Mealy machine with a smaller number of states. Thus, the task is to find a compromise between the size of the Mealy machine and prediction precision.

Objective function

For the evaluation we will use the objective function (Equation 4.4), which is defined as:

$$ObjValue_{pred} = \frac{correctly\ predicted\ bits}{length\ of\ the\ sequence}. \tag{4.4}$$

and returns a value in the range $[0\ldots1]$.

This objective function can be modified in order to take in to account the relation of the size of the Mealy machine with respect to the length of the bit mask.

4.3.2. Examples

11100 predictor

In this example, the environment is modeled based on the bit mask ’11100’. The length of the environment string is double the length of the mask. The number of states in Mealy machine is equal to 4, it is chosen to be less than the number of bits in the mask. The input and output alphabets are both $\{0, 1\}$.

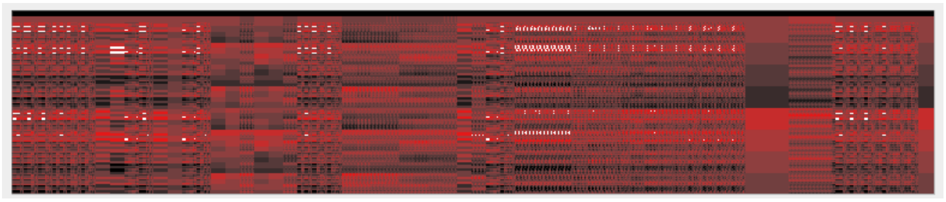


Figure 4.29 Search space landscape for $cSR_S(M_{11100})$ representation

Figure 4.29 illustrates the landscape of the search space. The best Mealy machine has the objective function value 1.0.

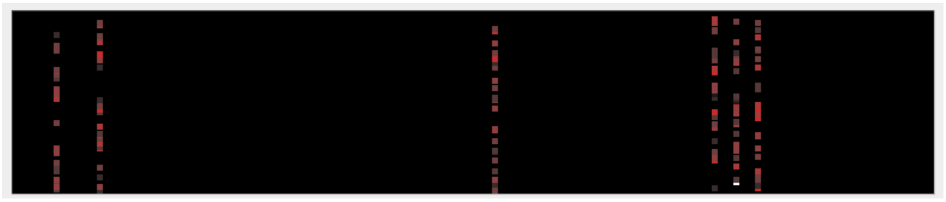


Figure 4.30 Set of explored points in the search space for $cSR_S(M_{11100})$ representation

The algorithm was able to find $cSR_S(M_{11100}) = \{[0,\ 1,\ 1,\ 2,\ 2,\ 3,\ 0\ ,\ 1], [1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 1]\}$, with the value 1.0, by only checking 167 points out of 1343488 during this run.

001111 predictor

In this example, the environment is modeled based on the bit mask ’001111’. The length of the environment string is double the length of the mask. The number of states in the Mealy machine is 4 states, as it is chosen to be less than the number of bits in mask. The input and output alphabets are both $\{0, 1\}$.

Figure 4.31 illustrates the landscape of the search space. The best Mealy machine in this context has the objective function value 1.0.

The algorithm was able to find $cSR_S(M_{001111}) = \{[1,\ 0,\ 1,\ 2,\ 3,\ 3,\ 1,\ 0], [0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1]\}$, with the value 1.0 by only checking 3684 points out of 1343488 during this run.

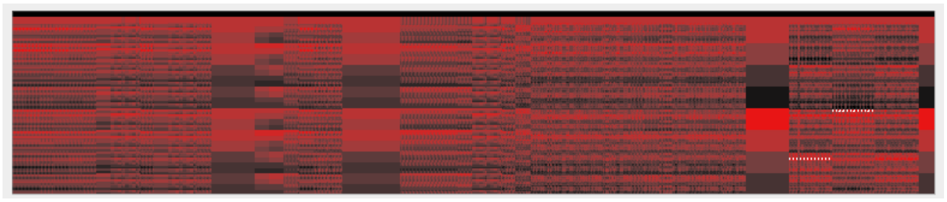


Figure 4.31 Search space landscape for $cSR_S(M_{001111})$ *representation*

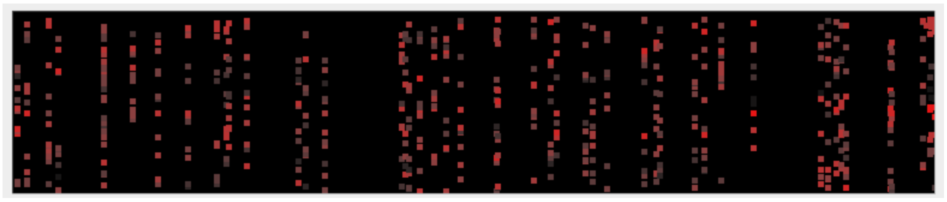


Figure 4.32 Set of explored points in the search space for $cSR_S(M_{001111})$ *representation*

4.4. Conclusion

This chapter includes discussion about certain tasks that can be solved by FSM identification, such as system identification, *artificial ant* and binary sequence predictor. Those examples illustrate how the proposed algorithm can be applied in situations that require usage of different FSM types (both Mealy and Moore machine), string representations and objective functions. Thus, the modular system allows to use similar methods for problems from different areas. In addition, the string representation unifies the search algorithm that is problem-independent.

The experimental results demonstrate that the canonical string representation helps significantly reduce the search space size. On the other hand, in the worst-case scenario, the two-level search algorithm leads to generating and evaluating a vast amount of unnecessary points.

CONCLUSIONS

FSMs are important class of models that can be utilized in various areas, such as modeling systems and agents, for representing grammars or hardware, etc. In general, the identification of FSMs is a problem relating to constructing FSM model from the examples of the behavior or description of the observed system. There are several tasks that can be solved by FSM identification including grammatical inference, reverse engineering, image processing, creating a model of behavior for agents.

Despite the fact that those tasks originate from different areas, there are several commonalities, for instance the behavior of the system is modeled by the FSM. Furthermore, the identification task assumes the use of the heuristic search due to the complexity of the deterministic procedure or because the behavior of the system is unknown. We propose to generalize those tasks and to omit the differences by defying the **modular system for** FSM **identification** (Chapter 1). This allowed us to separately treat the problem statement, search algorithm and solution evaluation process. Consequently, we can solve the above-mentioned tasks using the same method. For example this approach was benchmarked on the system identification, artificial ant and binary sequence predictor tasks (see Chapter 4). In addition, such modularity allows the use of different heuristic search methods or types of the FSM without any complex changes in the general search process, if required by the task.

Due to specifics of the FSM identification task, heuristic search algorithms are widely used. The most common choice is a variation of the Evolutionary Algorithm, e.g. GA or Genetic Programming. However, the family of population-based heuristic search methods is constantly growing. We propose new **Discrete Gravitationally Inspired Search Algorithm** (see Publication B and Section 3.5), which adapts the ideas of PSO and GSA. Both original algorithms were initially designed to work with a continuous search space, which is not applicable in our case. Yet, there are also modifications for the discrete and binary search spaces. For example, we applied a binary GSA for the FSM inference for system identification (see Publication C) and for some benchmarks it over-performed the traditional GA. The GSA and PSO algorithms have several inner mechanisms in common, which allows to create hybrid algorithms that employ the advantages of both methods. The issue with the traditional PSO is the complexity of control – there are three learning coefficients that need to be found before the search process. The proposed *Discrete Gravitationally Inspired*

Search Algorithm replaces those coefficient with computed values similar to the GSA (see Publication B), which in turn simplifies adjusting the search algorithm without reducing the exploration and exploitation abilities.

The performance of the heuristic search method strictly depends on the representation of solutions and the structure of the search space. There are two possibilities for the FSM representation – graph representation and string representation (see Chapter 2). In the first case, there is no decoding/encoding process, but the inner mechanisms of the heuristic search are more complex. For the string representations, the search algorithm works on discrete or binary strings (see Publication C and Subsection2.3.2). We focused solely on string representations, because there are more heuristic search methods that can be used with such representations. The most resource-consuming part of the heuristic search problem is not the algorithm itself nor the encoding/decoding process, but rather the evaluation of the solutions. Therefore, it is important to minimize the number of evaluations. As a solution we propose **canonical string representation** (see Section 2.7), which adapts the existing method for DFA enumeration developed by Almeida, Moreira and Reis ([46], [39] [47]) in the context of search space representation. First of all, this approach allows us to reduce the search space by removing isomorphic FSMs and FSMs with unreachable states. Secondly, such a canonical string representation helps to divide the search space into non-intersecting parts (see Section 3.2), which can be treated in parallel or sequentially. As regards sequential processing, we changed the search process to first process the parts with a higher probability of containing the solution. This helped significantly reduce the number of solution evaluations (see Subsection 4.2.3). However, in the worst-case scenario, if the first phase fails and all previous parts must be explored before the part containing the solution is found, the number of evaluations grows drastically. Thirdly, we can find the one-to-one correspondence between the FSM represented by canonical string and a triple of numbers, which can be employed as the key for the hash function which is used to store the already checked points of the search space. Storing the already checked points reduces the number of solution evaluations. Also, this 'key' (the triple of numbers) can be used as the main part of the visualization process. **Visualizing the search space** (see Subsection 3.2.7), naturally one with a reasonable size, helps to demonstrate the its structure and illustrate the behavior of the search algorithm.

The proposed approach was benchmarked on three well-known tasks (see Chapter 4): system identification (see Publication C), *artificial ant* problem (see Publication A) and binary sequence predictor.

Comparing the effectiveness of the proposed approach. Due to the various methods and algorithms for representation, the different heuristic search methods (Genetic Programming, Genetic Algorithm, Particle Swarm Optimization, etc.) and the different possible *evaluation function*s, the proposed method cannot be directly compared to already existing solutions. In addition, we minimized and structured the search space, so it is difficult to separate the effect of the proposed gravitationally-inspired search algorithm from the effect of search space minimization. However, as regards existing methods, the general structure of the solutions can traditionally be formulated as **problem statement + search algorithm + representation of** FSM.

The main contributions are:

1. **Unification** (one algorithm for different problems) – a modular system, which allows to fix the search algorithm and FSM representation and use one approach for different problem statements. To illustrate this capacity of the method, several experiments were conducted (see Chapter 4), which demonstrated the possibility of using one approach for different problems.

2. **The new string representation of** FSM, which is problem-independent and can be used with different search algorithms. Therefore, string representations can be compared without any knowledge of the problem and search method (Section 2.8). By presenting the new representation we removed all FSMs with unreachable states and isomorphic FSMs from search space, thus significantly reducing the search space .

3. **Two-stage algorithm and a new heuristic search algorithm** that is used for searching in search space partitions – the latter is discussed separately in publication B, where it was benchmarked and compared to traditional PSO and stochastic hill climbing, which are the closest methods.

The comparison 'GAs vs. binary GSAs' is made for the problem of system identification ($SR_B(FST)$ is used as representation of FSM) and the results are presented in Publication C. Those results can-not be directly compared to results discussed in Section 4.1, because it is impossible to separate the effect of the new representation from the effect of the new algorithm. Nevertheless, the number of total checked points can be observed.

Section 4.2 and Publication A also contain the experimental part, in which the problem statement is fixed. The proposed method is compared to other existing methods (Table 4.2).

Future work. There are several means of extending current research. First of all, we can adapt the proposed algorithm in order to solve other similar problems. For instance, grammatical inference or modeling of agents. Secondly, we can add a modification to the representation system in order to handle other types of machines, e.g. FAs or timed automata. Thirdly, the family of heuristic optimization techniques is constantly growing, so some of them can be modified to match the search space representation. Fourthly, the first phase of the search algorithm, namely selecting of the subspace that has a higher probability of containing the solution, must be improved to minimize the chances of the worst-case scenario occurring.

REFERENCES

[1] M. Brutscheck, B. Schmidt, M. Franke, A. T. Schwarzbacher, and S. Becker, "Identification of deterministic sequential finite state machines in unknown CMOS ICs," in *Signals and Systems Conference (ISSC 2009), IET Irish*, 2009, pp. 1–6.

[2] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, no. 3, pp. 302 – 320, 1978.

[3] P. Hingston, "A Genetic Algorithm for Regular Inference," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H. M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds. San Francisco, California, USA: Morgan Kaufmann, Jul. 2001, pp. 1299–1306.

[4] M. Tomita, "Dynamic construction of finite automata from examples using hill-climbing," in *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan, 1982, pp. 105–108.

[5] P. Kohli, "A new genetic algorithm based scheme for inferring finite state machines from accept/reject data samples." in *IICAI*, B. Prasad, Ed. IICAI, 2003, pp. 632–645.

[6] J. W. Horihan and Y.-H. Lu, "Improving FSM evolution with progressive fitness functions," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '04. New York, NY, USA: ACM, 2004, pp. 123–126.

[7] S. Lucas and T. J. Reynolds, "Learning deterministic finite automata with a smart state labeling evolutionary algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1063–1074, 2005.

[8] J. Bongard, H. Lipson, and S. Wrobel, "Active coevolutionary learning of deterministic finite automata," *Journal of Machine Learning Research*, vol. 6, p. 2005, 2005.

[9] M. M. Lankhorst, "A genetic algorithm for the induction of context-free grammars," 1993.

[10] S. M. Lucas, “Structuring chromosomes for context-free grammar evolution,” in *in Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE, 1994, pp. 130–135.

[11] A. Zomorodian, “Context-free language induction by evolution of deterministic push-down automata using genetic programming,” in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza, Eds. MIT, Cambridge, MA, USA: AAAI, 10–12 Nov. 1995, pp. 127–133.

[12] M. M. Lankhorst, “A genetic algorithm for the induction of nondeterministic pushdown automata,” University of Groningen, Tech. Rep., 1995.

[13] C. Manovit, C. Aporntewan, and P. Chongstitvatana, “Synthesis of synchronous sequential logic circuits from partial input/output sequences,” in *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, ser. ICES ’98. London, UK, UK: Springer-Verlag, 1998, pp. 98–105.

[14] P. Chongstitvatana and C. Aporntewan, “Improving correctness of finite-state machine synthesis from multiple partial input/output sequences,” in *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, 1999, pp. 262–266.

[15] L. Ngom, C. Baron, and J.-C. Geffroy, “Genetic simulation for finite state machine identification,” *Simulation Symposium, Annual*, vol. 0, p. 118, 1999.

[16] S. Tongchim and P. Chongstitvatana, “Parallel genetic algorithm for finite-state machine synthesis from input/output sequences,” 2000.

[17] N. Niparnan and P. Chongstitvatana, “An improved genetic algorithm for the inference of finite state machine,” in *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, vol. 7, oct. 2002, p. 5 pp. vol.7.

[18] X. Geng, “Solving identification problem for asynchronous finite state machines using genetic algorithms,” in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, ser. GECCO ’06. New York, NY, USA: ACM, 2006, pp. 1413–1414.

[19] H. Shayani and P. J. Bentley, “A more bio-plausible approach to the evolutionary inference of finite state machines,” in *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, ser. GECCO ’07. New York, NY, USA: ACM, 2007, pp. 2937–2944.

[20] A. Naidoo and N. Pillay, “The induction of finite transducers using genetic programming,” in *Proceedings of the 10th European conference on Genetic*

programming, ser. EuroGP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 371–380.

[21] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a theme in artificial life: The genesys/tracker system," in *C. G. Langten, C. Taylor, J. D. Farmer, and S. Rasmussen (Eds.), Artificial life II*. MA: Addison-Wesley, 1991, pp. 549–578.

[22] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[23] P. J. Angeline and J. B. Pollack, "Evolutionary module acquisition," in *Proceedings of the Second Annual Conference on Evolutionary Programming*, D. Fogel and W. Atmar, Eds., La Jolla, CA, USA, 25-26 Feb. 1993, pp. 154–163.

[24] P. J. Angeline and J. B. Pollack, "Coevolving high-level representations," in *Artificial Life III*, C. Langton, Ed. Reading MA: Addison-Wesley, 1994, pp. 55–71.

[25] I. Kuscu, "Evolving a generalised behavior: Artificial ant problem revisited," in *Seventh Annual Conference on Evolutionary Programming*, ser. LNCS, V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, Eds., vol. 1447. Mission Valley Marriott, San Diego, California, USA: Springer-Verlag, 25-27 Mar. 1998, pp. 799–.

[26] K. Chellapilla and D. Czarnecki, "A preliminary investigation into evolving modular finite state machines," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, vol. 2, 1999, pp. –1356 Vol. 2.

[27] D. A. Ashlock, *Evolutionary computation for modeling and optimization*. Springer, 2006.

[28] U. Cerruti, M. Giacobini, and P. Liardet, "Prediction of binary sequences by evolving finite state machines," in *Selected Papers from the 5th European Conference on Artificial Evolution*. London, UK, UK: Springer-Verlag, 2002, pp. 42–53.

[29] W. M. Spears and D. F. Gordon, "Evolving finite-state machine strategies for protecting resources," in *In Proceedings Of The International Symposium On Methodologies For Intelligent Systems 2000. 2000, Acm Special Interest Group On Artificial Intelligence*. Springer-Verlag, 2000, pp. 166–175.

[30] M. T. Tu, E. Wolff, and W. Lamersdorf, "Genetic algorithms for automated negotiations: a fsm-based application approach," in *Database and Expert Systems Applications, 2000. Proceedings. 11th International Workshop on*, 2000, pp. 1029–1033.

[31] S. Lucas and T. Reynolds, “Learning finite-state transducers: Evolution versus heuristic state merging,” *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 3, pp. 308–325, 2007.

[32] K. Benson, “Evolving finite state machines with embedded genetic programming for automatic target detection,” in *Proceedings of the 2000 Congress on Evolutionary Computation*, vol. 2, 2000, pp. 1543–1549 vol.2.

[33] S. M. Lucas, “Evolving finite state transducers: Some initial explorations,” in *In European Conference on Genetic Programming, EuroGP 2003*. Springer, 2003, pp. 130–141.

[34] V. Fabera, V. Janes, and M. Janesova, “Automata construct with genetic algorithm,” in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006, pp. 460–463.

[35] P. Petrovic, “Incremental evolutionary methods for automatic programming of robot controllers,” Ph.D. dissertation, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, 2007.

[36] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, Nov 1995, pp. 1942–1948.

[37] D. Angluin and C. H. Smith, “Inductive inference: Theory and methods,” *ACM Comput. Surv.*, vol. 15, pp. 237–269, September 1983.

[38] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley, 1979.

[39] M. Almeida, M. Moreira, and R. Reis, “Enumeration and generation with a string automata representation,” *THEORET. COMPUT. SCI.*, p. 2007, 2007.

[40] L. Fogel, A. Owens, and M. Walsh, *Artificial intelligence through simulated evolution*. Wiley, 1966.

[41] M. Spichakova, “Genetic Inference of Finite State Machines,” Master’s thesis, Tallinn University of Technology, Tallinn, Estonia, 2007.

[42] M. Spichakova, “An approach to inference of finite state machines based on a gravitationally-inspired search algorithm,” *Proceedings of the Estonian Academy of Sciences*, vol. 62, pp. 39–46, 2013.

[43] N. Niparnan, “A genetic algorithm for finite state machine inference,” 2002.

[44] S. M. Lucas and T. J. Reynolds, "Learning DFA: evolution versus evidence driven state merging," in *Evolutionary Computation, 2003. CEC 03. The 2003 Congress on*, vol. 1, 2003, pp. 351–358 Vol.1.

[45] L. D. Chambers, *Practical Handbook of Genetic Algorithms*. Boca Raton, FL, USA: CRC Press, Inc., 1995.

[46] R. Reis, N. Moreira, and M. Almeida, "On the representation of finite automata," in *Proc. of DCFS05*, 2005, pp. 269–276.

[47] M. Almeida, N. Moreira, and R. Reis, "Aspects of enumeration and generation with a string automata representation," *CoRR*, vol. abs/0906.3853, 2009.

[48] R. Formato, "Central force optimization: a new metaheuristic with applications in applied electromagnetics," *Progress in Electromagnetics Research*, vol. PIER 77, pp. 425–491, 2007.

[49] Y.-T. Hsiao, C.-L. Chuang, J. J.-A., and C. C.-C., "A novel optimization algorithm: space gravitational optimization," in *Systems, Man and Cybernetics, 2005 IEEE International Conference*, vol. 3, 2005, pp. 2323–2328.

[50] B. Webster and P. J. Bernhard, "A local search optimization algorithm based on natural principles of gravitation," Florida Institute of Technology, Tech. Rep. CS-2003-10, 2003.

[51] B. Webster, "Solving combinatorial optimization problems using a new algorithm based on gravitational attraction," Ph.D. dissertation, Florida Institute of Technology, Melbourne, FL, USA, 2004.

[52] E. Rashedi, H. Nezamabadi-pour, S. Saryazdi, and M. M. Farsangi, "Allocation of static var compensator using gravitational search algorithm," in *First Joint Congress on Fuzzy and Intelligent Systems Ferdowsi University of Mashhad*, Iran, August, 29-31 2007, pp. 29–31.

[53] E. Rashedi, H. Nezamabadi-pour, and S. Saryazdi, "GSA: A gravitational search algorithm," *Inform. Sciences*, vol. 179, no. 13, pp. 2232–2248, 2009.

[54] E. Rashedi, H. Nezamabadi-pour, and S. Saryazdi, "Filter modeling using gravitational search algorithm," *Eng. Appl. Artif. Intell.*, vol. 24, pp. 117–122, February 2011.

[55] B. Zibanezhad, K. Zamanifar, N. Nematbakhsh, and F. Mardukhi, "An approach for web services composition based on qos and gravitational search algorithm," in *Proceedings of the 6th International Conference on Innovations in Information Technology*, ser. IIT'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 121–125.

[56] R. Povinelli, “Comparing genetic algorithms computational performance improvement techniques,” in *Artificial Neural Networks in Engineering*, 2000, pp. 305–310.

[57] R. Povinelli and X. Feng, “Improving genetic algorithms performance by hashing fitness values,” in *Artificial Neural Networks in Engineering*, 1999, pp. 399–404.

[58] M. Spichakova, “Modified particle swarm optimization algorithm based on gravitational field interactions,” *Proceedings of the Estonian Academy of Sciences*, vol. 65, pp. 15–27, 2016.

[59] L. Pitt and M. K. Warmuth, “The minimum consistent dfa problem cannot be approximated within any polynomial,” *J. ACM*, vol. 40, no. 1, pp. 95–142, Jan. 1993.

[60] P. Dupont, L. Miclet, and E. Vidal, “What is the search space of the regular inference?” in *Proceedings of the Second International Colloquium on Grammatical Inference (ICGI’94)*. Springer Verlag, 1994, pp. 25–37.

[61] A. Oliveira, J. M. Silva, and V. Honavar, “Efficient algorithms for the inference of minimum size DFAs,” in *Machine Learning*. Springer, 2000, p. 2001.

[62] H. Sugiura, T. Mizuno, and E. Kita, “Santa fe trail problem solution using grammatical evolution,” in *2012 International Conference on Industrial and Intelligent Information (ICIII 2012)*, 2012, pp. 36–40.

[63] D. Chivilikhin, V. Ulyantsev, and A. Shalyto, “Solving five instances of the artificial ant problem with ant colony optimization,” in *Proceedings of the 7th IFAC Conference on Manufacturing Modelling, Management, and Control*, 2013, pp. 1043–1048.

[64] S. Christensen and F. Oppacher, “Solving the artificial ant on the Santa Fe trail problem in 20,696 fitness evaluations.” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO 07*, 2007, p. 1574–1579.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Jaan Penjam for being a real ‘academic father’ to me. He demonstrated endless patience during all these years and finally motivated me when I was on the verge of giving up.

I am grateful to my colleagues at the Department of Software Science for their professional help, and personal motivation and support.

Special thanks go to Jelena Sanko for scientific discussions and insightful comments related to this work.

The work reported in this thesis was supported by the Estonian Ministry of Research and Education institutional research grant no. IUT33-13, by the Estonian Doctoral School in Information and Communication Technology (IKTDK), by the ERDF through the ITC project MBJSDT and Estonian national CoE project EXCS, and by the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

ABSTRACT
Discrete Gravitational Swarm Optimization Algorithm for System Identification

We presented a method for the identification of FSMs, which is based on a heuristic optimization algorithm.

The considered method is constructed as the *modular system*, which separately theats the problem statement, the representation of the FSM and the search algorithm. This allows to adapt the proposed search algorithm to different problems. We discussed 'system identification', 'artificial ant problem' and 'binary string predictor' test scenarios, however the set of the problems is wider.

The special coding system *'Canonical String Representation'* is used to represent the search space. Proposed string representation of FSMs entails the adaptation of the existing method for enumeration of FAs (see Section 2.7). It was updated to take into account the FST output function. Firstly, such representation allows to *minimize the search space*. Secondly, the search space can be partitioned and non-intersecting partitions can be considered either in *parallel* or sequentially in time.

The search algorithm is *problem-independent* and is based on a combination of PSO and GSA. Moreover, the specifics of the search space representation provides the possibility of creating *two search stages* — the search for the 'best' partition (meta-search) and the search inside the partition.

The partition-local search is based on the ideas of PSO, but it was modified so as to be able to work in a discrete search space, because our search space is presented by decimal strings. Thus, all of the operators of the algorithm were redesigned. In addition, due to the complex structure of the search space, the standard PSO is not performing well. Hence, we propose adapting the ideas of the modern gravitational algorithm and presented a new hybrid *'Discrete Gravitational Swarm Optimization algorithm'*.

KOKKUVÕTE
Diskreetne gravitatsioonilist vastasmõju arvestav osakeste parvega optimeerimise meetod süsteemide identifitseerimiseks

Mudeli identifitseerimine võimaldab tuletada süsteemi sisemise kirjelduse tema väliselt jälgitava käitumise põhjal. Süsteemide kirjeldamiseks kasutatakse tihti lõplikke olekumasinaid. Probleemiks on asjaolu, et lõpliku olekumasina tuletamine sisend-väljundpaaride näidete alusel on NP-keeruline ülesanne. Keerukuse vähendamiseks võib kasutada heuristilisi meetodeid.

Käesoleva töö ülesandeks oli välja töötada algoritm lõpliku olekumasina leidmiseks kasutades stohhastilise optimeerimise meetodeid, keskenduses kahele lõplikule olekumasina põhitüübile: Moore ja Mealy masinale. Nende jaoks on töös leitud uus gravitatsiooniseadusest inspireeritud otsimisalgoritm meetodid etteantud olekute arvuga masinate genereerimiseks sisendi–väljundi kirjeldusest.

Töös antakse lühike ülevaade lõplike olekumasinate teooriast, esitatakse formaalsete keelte ja lõplike automaatide teoreetilised alused, kirjeldatakse väljundiga lõplikke olekumasinaid — Mealy ja Moore masinaid. Töös käsitletakse loodid algoritmi rakendamiseks vajalikku masinate esitust ja sellega kaasnevaid probleeme (vigased indiviidid, saavutamatud olekud ja vigased üleminekud). Esitatakse uus kanooniline lõpliku olekumasina kodeerimisviis numbrijadana, mis lahendab neid probleeme ja võimaldab ka visualiseerida ning analüüsida otsimisruumi ja algoritmi käitumist kahemõõtmelisel juhul.

Töös kirjeldatakse stohhastilise algoritmide põhiprintsiipe ning esitatakse osakeste parve optimeerimisalgoritmi modifikatsioon DGSO, kus otsimismeetodi parameetrid arvutatakse osakestevahelise gravitatsioonilise vastasmõju põhjal. DGSO algoritmi võib käsitleda kui hübriidi osakeste parve optimeerimise (PSO, i.k. Particle Swarm Optimization) ja gravitatsioonilise optimeerimise meetodist, kus osakeste liikumistrajektoorid arvutatakse sarnaselt punktmasside liikumisvõrranditele gravitatsiooniväljas.

Algoritmi töö demonstreerimiseks kirjeldatakse töös ka lõplike olekumasinate genereerimise prototüüpi. Testimiseks tehtud eksperimentidest antakse ülevaade: iga ülesande korral kirjeldatakse selle püstitust ja tulemusi, kirjeldatakse funktsioone, mis võimaldavad hinnata konkreetse olekumasina sobivust mudeliks ja algoritmi juhtimiseks kasutatavad parameetrid. Eksperimendid näitavad, et lähenemisel on arvestatav potentsiaal.

Publication A

Spichakova, Margarita (2017). ***Gravitationally Inspired Search Algorithm for Solving Agent Tasks.*** *Baltic Journal of Modern Computing, 5(1), 87 – 106.*

Baltic J. Modern Computing, Vol. 5 (2017), No. 1, pp. 87–106
http://dx.doi.org/10.22364/bjmc.2017.5.1.06

Gravitationally Inspired Search Algorithm for Solving Agent Tasks

Margarita SPICHAKOVA

Institute of Software Science at Tallinn University of Technology
Tallinn, Estonia

margarita.spitsakova@ttu.ee

Abstract. Artificial ant problem is defined as constructing the agent that models the behavior of the ant on trail with food. The goal of the ant is to eat all food on trail with limited number of steps. Traditionally, the recurrent neural network or state machines are used for modeling ant and heuristic optimization methods for example Genetic Programming as search algorithm. In this article we use Mealy machines as ant model in combination with Particle Swarm Optimization method and heuristic algorithms inspired by gravity. We propose new gravitationally inspired search algorithm and its application to artificial ant problem. Proposed search algorithm requires discrete search space, so the specific string representation of Mealy machine is introduced. The simulation results and analysis of the search space complexity show that the proposed method can reduce the size of the search space and effectively solve the problem.

Keywords: Soft computing, swarm intelligence, agent, artificial ant, gravitational search, Particle Swarm Optimization.

Introduction

Agent games on grid world are used to understand the behavior of agents, especially their controllers in real world tasks. To behave correctly on the grid, the controller requires some memory, so, for example, *finite state machine* can be used to model agent.

The *artificial ant* task can be described as designing trail tracker, which acts as *artificial ant* and follows some trail, which contains food (see Fig. 1). The goal of ant is to collect maximal amount of food for limited number of steps, traditionally 200 steps. In our case, ant is modeled by *finite state machine* (Mealy type) like agent, but it is possible to use other models, for example recurrent neural networks. The input of such machine is only one variable: is there food in the next cell, with values `FOOD` and `EMPTY`, the outputs are defined as actions of ant: `WAIT`, `TURN LEFT`, `TURN RIGHT`, `MOVE`.

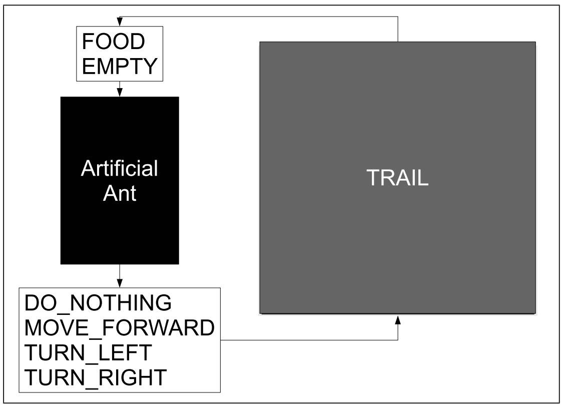


Fig. 1. General model of *artificial ant*

There is no deterministic solution for this problem, so stochastic optimization methods can be used for solving *artificial ant* problem. In most of the cases the population based heuristic methods, such as *Evolutionary Algorithm*s or simulated annealing, are used.

Initially, this task was proposed by Jefferson (1991) to benchmark *Evolutionary Algorithm*s and was researched afterward by many authors: Koza (1992), Angeline (1993), (1994), Kuscu (1998). We use *artificial ant* task for benchmarking our proposed method based on gravitationally inspired search algorithm.

This task is applicable not only in case of *FSM* inference, but also in a case of *Genetic Programming* by Koza (1992) and artificial neural network learning by Jefferson (1991). Chellapilla (1999) uses modular *Mealy machine* as *artificial ant* and *Evolutionary Programming* procedure for the optimization. We focus only on traditional *Mealy machine*s.

In this article we propose a new stochastic optimization method, which can be applied for the problem of *FSM* identification. The proposed method is inspired by combination of *Particle Swarm Optimization* (*PSO*) method and *Gravitational Search Algorithm* (*GSA*).

The article is constructed as follows: Section 1 describes the problem statement in details, Section 3 presents the general idea of proposed stochastic optimization method, Section 4 covers the problem of *FSM* representation, Section 5 gives the main ideas of the method with application to *FSM* identification and Section 6 presents the simulation results and analysis.

1 Artificial ant problem

The problem in hand is known as '*Artificial ant* problem' or 'Trail tracker problem'. It was originally proposed by Jefferson et al. (1991). The goal is construct the agent, which takes information about the food in the next cell and moves on the grid, so that it finds the optimal trail to collect all food on the grid with pre-given amount of steps.

1.1 Trail

The grid is presented by 32×32 toroidal structure, so that if ant is at the bottom of such grid and the next move is `MOVE DOWN`, the next cell will be at the top.

The food on the grid is located in a special way, constructed to make the task more complex. The are two well known trails *John Muir Trail* (Fig. 2 (left)) developed by Jefferson et al. (1991) and *Santa Fe Trail* (Fig. 2 (right)). Both of them contain 89 cell of food.

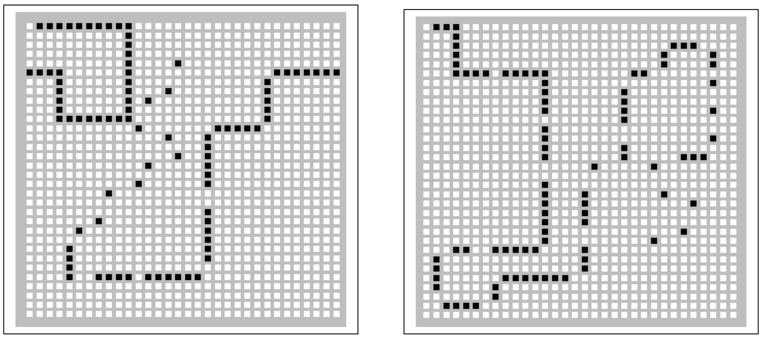


Fig. 2. *John Muir Trail* (left) and *Santa Fe Trail* (right)

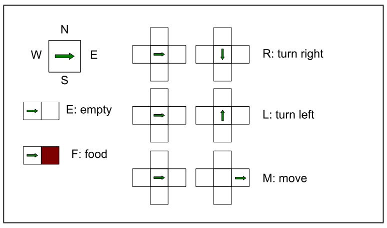
1.2 *Finite state machine* as ant model

The ant is modeled by Mealy type machine with n states. The ant sees only one cell directly in front of him, so there is only one input variable with two values (events). So, the input alphabet contains only two chars: `F (FOOD)` – there is food in the next cell, `E (EMPTY)` – the next food is empty.

The allowed actions of ant can be coded by 4 letter alphabet: `W (WAIT)` – the ant will do nothing at this step, `M (MOVE)` – the ant will move to the next cell, if there is a food it will eat it (the eaten food is removed from the grid), `R (TURN RIGHT)` – the ant will stay in the same cell, but turn the right, `L (TURN LEFT)` – the ant will stay in the same cell, but turn the left. WAIT action can be omitted, in that case the size of the alphabet is 3.

The ant can see only the cell in front of it, the address of the cell is defined by orientation of the ant: N (NORTH) – ant is looking the top cell, E (EAST) – ant is looking the right cell, S (SOUTH) – ant is looking the bottom cell, W (WEST) – ant is looking the left cell.

The start position of ant $(0, 0)$ is top left cell. The initial orientation is EAST. So, we define $\Sigma = \{E, F\}$ and $\Delta = \{W, M, L, R\}$. Fig. 3 (left) demonstrates the defined alphabets (actions and events) for the situation, where ant is oriented EAST.



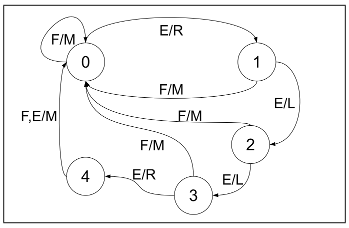


Fig. 3. Ant actions (left) and *Mealy machine* as *artificial ant* (right)

The *FSM*, which is able to fulfill the task is quite complex (we measure complexity by number of states). Jefferson (1991), the author of original task, constructed the *FSM* with 5 states (Fig. 3 (right)), which was able to eat 81 pieces of food out of 89 on *John Muir Trail* by 200 steps and eat all food by 314 steps. As you can see, this ant with 5 steps was not able to eat all the food by 200 steps, to do so, the number of states (memory) must be greater. In the literature ant with up to 13 states are used.

1.3 Ant evaluation

If we want to answer the question 'How well the given ant solves the problem?', we need to construct so called *objective function*. The score of the ant can be found only by modeling situation. The process of *objective function* computation is the most time consuming part of the search algorithm.

There are several possible function. In the easiest case, we can take into account only amount of food that was eaten during given number of steps. We modify this function so that it returns value $\in [0 \ldots 1]$ (Equation 1): we divide the number of eaten food by the total number of food on grid.

$$ObjValue = \frac{eaten\ food}{total\ food\ on\ trail} \tag{1}$$

As you have noticed, presented function does not take into account the number of states of the model and the number of steps that was required to eat all the food, if it is less than given number of steps. The *objective function* can be modified to minimize the number of states, but we use the simplified version.

2 Inference of finite state machines

To define our problem in the context of stochastic optimization methods we need first of all define several concepts. We have the *problem statement* – 'find the *Mealy machine* with n states, which models the *artificial ant* that eats all food on given trail in given number of steps'. In fact we have our *search space* defined as set of all possible *Mealy machine*s with n states. Each point of such space is *candidate solution*. For each *candidate solution* we can assign *objective value* using the above defined *objective function* (Equation 1). Now, we can redefine the problem as **find the candidate solution with maximal objective value**, which is exactly the definition of the optimization task.

Such tasks are solvable by population based heuristic algorithms.

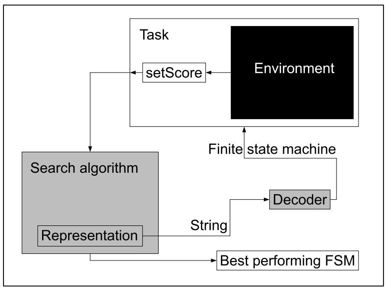


Fig. 4. The outline of the *FSM* search process

The proposed system implements the ideas of unified methods for *FSM* identification consists of three main modules (see Fig. 4). Each module of the system presents the independent part of the system and can be replaced by another implementation or definition:

– *'Task' module* contains the implementation of basic concepts and definitions that are used to describe the statement of the problem. To describe the problem, we need to define the type of the *FSM*, choose alphabets, number of states and construct the *FSM* evaluation algorithm. These parameters (number of states, type of the machine and alphabets) can be derived from problem specification (see Section 1).
– *'Search algorithm' module* contains the implementation of different stochastic optimization algorithms, such as *GA* or *PSO* (see Section 5).
– *'Representation+Decoder' module.* This subsystem contains algorithms for definition of *FSM* and string representation of *FSM* (see Section 4).

3 Heuristic search

There are many different heuristics, for example *Genetic Algorithm*s (*GA*) or *Particle Swarm Optimization* (*PSO*). Some of them are inspired by natural mechanisms, such as evolutionary theory, physics or social behavior.

But still, they have several mechanisms in common. First of all, initial set of the *candidate solution*s are generated randomly. Each of them is evaluated and the *score value* is assigned. Using those values some of the *candidate solution*s are chosen for modification. There are several algorithm specific modification methods. Using them we can improve the *candidate solution*s. After the modifications all *candidate solution*s are evaluated again. The process continues until the optimal solution is found or the *number of generation* is reached. The important fact is that in most of the cases *candidate solution*s are usually presented indirectly, encoded to some structure: Boolean vector, vector of integer or real values. The choice of *representation* and search algorithm plays important role in resolvability of the problem.

3.1 *Evolutionary Algorithm*s

Evolutionary Algorithm (*EA*) is a search algorithm based on simulation of biological evolution. In *EA* some mechanisms inspired by evolution are used: reproduction, mutation, recombination, selection. *Candidate solution* presents the individuals of the evolution.

Some differences in implementation of those principles characterize the instances of *EA* (Bäck 1997):

– *Genetic Algorithm*s (originally described by Holland in 1962) (*GA*) — the solution represented as an array of numbers (usually binary numbers). A recombination operator is also used to produce new solutions.
– *Evolutionary strategies* (developed by Rechenberg and Schwefel in 1965) — use the real number vectors for representing solution. Mutation and crossover are essential operators for searching in search space.
– *Evolutionary Programming*s (developed by Lawrence J. Fogel in 1962) — the algorithm was originally developed to evolve *FSM*, but most applications are for search spaces involving real-valued vectors. Does not incorporate the recombination of individuals and emphasizes the mutation.

*EA*s are often used to solve *artificial ant* task: Jefferson (1991), Koza (1992), Angeline (1993), (1994), Kuscu (1998).

3.2 Standard *Particle Swarm Optimization* algorithm

Particle Swarm Optimization algorithm is inspired by social behavior of some set of objects, for example bird flock or fish school. Initially, it is designed by Kennedy at al. (1995) for the real-valued vector.

There is the set of the particles – *swarm*. Each of them is characterized by *position vector*, *velocity vector* and the best known position for this object. Also, there is the *global best known position* for the whole *swarm*.

The *position* vector $p_d, d \in [0 \ldots n]$ presents the *candidate solution*. Dimensionality n of the vector depends on the problem size.

We assign value for each *candidate solution* using *evaluation function*. According to those values we can choose the global best known position $Gbest$, which is the point with optimal value found so far by the whole swarm, and the local best known position $Pbest$, which is the best position that was found by this exact particle.

The *velocity* vector $v_d, d \in [0 \ldots n]$ represents the trend of movement of the particle. It is computed by using (2),

$$v_d(t) = \alpha \cdot v_d(t-1) + \beta \cdot r_1 \cdot (Pbest_d - p_d(t-1)) + \gamma \cdot r_2 \cdot (Gbest_d - p_d(t-1)) \quad (2)$$

where:

- $\alpha,\ \beta,\ \gamma$ are learning coefficients and α represents the inertia, β - the cognitive memory, γ - the social memory. Those coefficients must be defined by user.
- r_1 and r_2 are random values in range $[0 \ldots 1]$.
- $Pbest_d$ is a local best known position for this particle, $Gbest_d$ is global best known position of the *swarm*.
- $v_d(t)$ is the new value of the velocity vector at dimension d, $v_d(t-1)$ is the previous value of the velocity.

The new position $p_d(t)$ is simply defined as sum of the previous position $p_d(t-1)$ and new velocity $v_d(t)$ (3).

$$p_d(t) = p_d(t-1) + v_d(t) \quad (3)$$

3.2.1 *PSO* algorithm work-flow The initialization part consists of defining required learning parameters, setting up boundaries of the search space and generating the swarm with random position and velocity. The search process is iterative update of the positions and velocities. The process ends when the ending criteria are met: either the number of iterations is exceeded or the optimal solution is found.

3.3 Gravity as inspiration for optimization algorithms

There are four main forces acting in our universe: gravitational, electromagnetic, weak-nuclear and strong nuclear. These forces define the way our universe behaves and appears. The weakest force is gravitational, it defines how objects move depending on their masses.

The gravitational force between two objects i and j is directly proportional to the product of their masses and inversely proportional to square distance between them

$$F_{ij} = G\frac{M_j \cdot M_i}{R_{ij}^2}. \quad (4)$$

Knowing the force acting on body we can compute acceleration as

$$a_i = \frac{F_i}{M_i}. \quad (5)$$

To construct the search algorithm based on gravity, we can adapt the following ideas:

- Each object in the universe has mass and position.
- There are interactions between objects, which can be described by using law of gravity.
- Bigger objects (with greater mass) create larger gravitational field and attract smaller ones.

During the last decade some researchers have tried to adapt the idea of gravity to find out optimal search algorithms. Such algorithms have some general ideas in common:

- The system is modeled by objects with mass.
- The position of those objects describes the solution and the mass of the objects depends on the *evaluation function*.
- The objects interact with each other using gravitational force.
- The objects with greater mass present the points in search space with better solution.

Using these characteristics, it is possible to define family of optimization algorithms based on gravitational force. For example, *Central Force Optimization (CFO)* is deterministic gravity based search algorithm proposed and developed by Formato (2007). It simulates the group of probes which fly into search space and explore it. Another algorithm, *Space Gravitational Optimization (SGO)* was developed by Hsiao and Chuang (2005). It simulates asteroids flying through curved search space. A gravitationally-inspired variation of local search algorithm, *Gravitational Emulation Local Search Algorithm (GELS)* was proposed by Webster (2003), (2004). Another one, *Gravitational Search Algorithm* (*GSA*) was proposed by Rashedi (2009)as a stochastic variation of *CFO*.

Basically, the gravitationally inspired algorithms are quite similar to *PSO* algorithms. Instead of particle swarm we have set of bodies with masses, ideas of position and velocity vectors are the same, the movement laws are similar.

The idea of hybridization of *PSO* algorithm and gravitationally inspired search algorithms is not new. There are several existing algorithms that adapt both ideas (*PSO* algorithm and gravity) to construct heuristic search algorithm: *PSOGSA* - *PSO*algorithm and *Gravitational Search Algorithm* developed by Mirjalili (2010), Extended *Particle Swarm Optimization* Algorithm Based On Self - Organization Topology Driven By Fitness - *PSO* algorithm and Artificial Physics, created by Mo et al. (2011), Gravitational Particle Swarm, proposed by Tsai (2013), Self-organizing *Particle Swarm Optimization* based on Gravitation Field Model, created by Qi et al. (2007).

The traditional way of hybridization of *PSO* algorithm with gravitationally-inspired search algorithms is to add gravitational component to the velocity computation. (2) has additional component, which is computed by using gravitational interactions. Unfortunately, this makes the behavior of the search algorithm even more complex and unpredictable. Additionally, the user defined parameters still need to be found.

The choice of the optimization method and its effectiveness strictly depends on type of the search space. In next Section we discuss the definition of our search space.

4 Representation of finite state machines

Let's have a target *Mealy machine* with n states.

- Input alphabet: $\Sigma = \{i_0, \ldots, i_{k-1}\}$;
- Output alphabet: $\Delta = \{o_0, \ldots, o_{m-1}\}$;
- Set of states: $Q = \{q_0, \ldots, q_{n-1}\}$;

One row of the transition table for such *Mealy machine* can be described by structure presented on Fig. 5, where we store the information for corresponding state q_j: for all transition from this state with respect to input symbol $i_{0\ldots k-1}$ we encode output value $o^{i_{\cdots}}$ and the label of target state $q^{i_{\cdots}}$.

State q_j					
o^{i_0}	q^{i_0}	$o^{i_{\cdots}}$	$q^{i_{\cdots}}$	$o^{i_{k-1}}$	$q^{i_{k-1}}$

Fig. 5. One row of *Mealy machine* transition table

The structure required to code whole transition table is constructed as concatenation of such sections in the order of state labels (see Fig. 6).

State q_0					$\ldots$	State q_{n-1}				
$o_0^{i_0}$	$q_0^{i_0}$	$\ldots$	$o_0^{i_{k-1}}$	$q_0^{i_{k-1}}$	$\ldots$	$o_{n-1}^{i_0}$	$q_{n-1}^{i_0}$	$\ldots$	$o_{n-1}^{i_{k-1}}$	$q_{n-1}^{i_{k-1}}$

Fig. 6. $\mathrm{SR(MeFST)}$ as concatenation of transition table rows

Definition 1 ($\mathrm{SR(MeFST)}$**: String representation of *Mealy machine*).** The string representation of *Mealy machine* is a structure in form:
$o_0^{i_0} q_0^{i_0} \ldots o_0^{i_{k-1}} q_0^{i_{k-1}} \ldots o_{n-1}^{i_0} q_{n-1}^{i_0} \ldots o_{n-1}^{i_{k-1}} q_{n-1}^{i_{k-1}}$, where $[o_0^{i_0} \ldots o_{n-1}^{i_{k-1}}] \in [0 \ldots m-1]$ presenting codes for output values of the transitions and $[q_0^{i_0} \ldots q_{n-1}^{i_{k-1}}] \in [0 \ldots n-1]$ presenting target states of the transitions.

Theorem 1 (The space complexity of $\mathrm{SR(MeFST)}$**).** *The length of* $\mathrm{SR(MeFST)}$ *representing* Mealy machine *with n states and over input alphabet k with output alphabet with m symbols is*

$$((1+1) \times k) \times n$$

The number of corresponding $\mathrm{SR(MeFST)}$ *strings is*

$$((m \times n)^k)^n$$

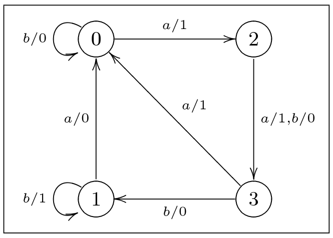


Fig. 7. Transition diagram of *Mealy machine* M_{me1}

Example 1. Let's take a look at a *Mealy machine* M_{me1} with transition diagram represented on Fig. 7. $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$.

The string representation for such machine will be
$SR(M_{me1}) = [1, 2, 0, 0, 0, 0, 1, 1, 1, 3, 0, 3, 1, 0, 0, 1]$ (see Fig. 8).

State 0				State 1				State 2				State 3			
a		b		a		b		a		b		a		b	
1	2	0	0	0	0	1	1	1	3	0	3	1	0	0	1

Fig. 8. String representation $SR(M_{me1})$

The separation of the the transition and output functions can be easily done by constructing two strings from a SR(MeFST) string. Fig. 9 describes separating the transition and output functions transformation for *Mealy machine*.

State q_0					...	State q_{n-1}				
$o_0^{i_0}$	$q_0^{i_0}$	...	$o_0^{i_{k-1}}$	$q_0^{i_{k-1}}$	...	$o_{n-1}^{i_0}$	$q_{n-1}^{i_0}$	...	$o_{n-1}^{i_{k-1}}$	$q_{n-1}^{i_{k-1}}$

State q_0			...	State q_{n-1}		
$q_0^{i_0}$	...	$q_0^{i_{k-1}}$	...	$q_{n-1}^{i_0}$	...	$q_{n-1}^{i_{k-1}}$

State q_0			...	State q_{n-1}		
$o_0^{i_0}$	...	$o_0^{i_{k-1}}$	...	$o_{n-1}^{i_0}$	...	$o_{n-1}^{i_{k-1}}$

Fig. 9. $\mathrm{SR_S}(\mathrm{MeFST})$: Separating transition $SR_S(MeFST.transition)$ (right) and output functions $SR_S(MeFST.output)$ (left) for *Mealy machine* SR(MeFST) (top)

Based on this transformation process the SR(MeFST) is defined (Definition 2).

Definition 2 ($\mathrm{SR_S}(\mathrm{MeFST})$**: Separated string representation of *Mealy machine*).** The separated string representation of *Mealy machine* is a structure formed from

SR(MeFST):

$$SR_S(MeFST) = \{SR_S(MeFST.transition), SR_S(MeFST.output)\},$$

where

- $SR_S(MeFST.transition)$ presents transition function $q_0^{i_0} \dots q_0^{i_{k-1}} \dots q_{n-1}^{i_0} \dots q_{n-1}^{i_{k-1}}$, with $[q^{i_0} \dots q_{n-1}^{i_{k-1}}] \in [0 \dots n-1]$ presenting target states of the transitions and
- $SR_S(MeFST.output)$ presents output function $o_0^{i_0} \dots o_0^{i_{k-1}} \dots o_{n-1}^{i_0} \dots o_{n-1}^{i_{k-1}}$, where $[o^0 \dots o^{n-1}] \in [0 \dots m-1]$ is presenting codes for output values.

Example 2. Let's return to M_{me1} (Example 1). The original code is
$SR(M_{me1}) = [1, 2, 0, 0, 0, 0, 1, 1, 1, 3, 0, 3, 1, 0, 0, 1]$,
so if we transform it to separated string representation the result will be:
$SR_S(M_{me1}) = \{[2, 0, 0, 1, 3, 0, 3, 0, 0, 1], [1, 0, 0, 1, 1, 0, 1, 0,]\}$

Example 3. The ant on Fig. 3 (right) can be presented by
$cSR_S(M_{ant81t314}) = \{[1,\ 0,\ 2,\ 0,\ 3,\ 0,\ 4,\ 0,\ 0,\ 0],\ [3,\ 1,\ 2,\ 1,\ 2,\ 1,\ 3,\ 1,\ 1,\ 1]\}$.

The $SR_S(FSM.transition)$ code depends on the labels of the states and their ordering, renaming states leads to isomorphisms. We can solve this problem by determining the way the state labels will be named. To do so, we adapt a method known as *normal form string*.

4.1 Normal form strings

We describe the basic theory we use in further algorithms. The used methods were proposed by Almeida, Moreira and Reis (2005), (2007), (2009) in a context of enumeration of deterministic finite state acceptors (*DFA*). We present only some definitions and algorithms we require for the research, so more information, proofs and other algorithms can be found in original sources.

The main goal of their approach is to find unique string representation of initially connected (all states are reachable from the initial one) *DFA* (*ICDFA*), this is done by construction of state label ordering. This representation is unique.

Suppose we have $ICDFA_\emptyset = (Q, \Sigma, \delta, q_0)$, where Q is a set of states $|Q| = n$, q_0 is initial states, Σ is the input alphabet with k symbols and δ is a transition function. As you can notice, the set of final states is omitted.

The *canonical string representation* (based on canonical order of states of the *ICDFA*) is constructed by exploring set of states of given *ICDFA* using bread-first search by choosing outgoing edges in the order of symbols in Σ.

So, first of all, the ordering of input alphabet must be defined, for given $\Sigma = \{i_0, i_1, \dots, i_{k-1}\}$, there is order $i_0 < i_1 < \dots < i_{k-1}$, for example the lexicographical ordering can be used.

For given $ICDFA_\emptyset$, the representing string will be in form $(s_j)_{j \in [0 \dots kn-1]}$ with $s_j \in [0 \dots n-1]$ and $s_j = \delta(\lfloor j/k \rfloor, i_{j \bmod k})$. There is a one-to-one mapping between string $(s_j)_{j\in[0\ldots kn-1]}$ with $s_j \in [0\ldots n-1]$ satisfying rules (more details in Almeida (2009) work):

$$(\forall m \in [2\ldots n-1])(\forall j \in [0\ldots kn-1])$$
$$(s_j = m \Rightarrow (\exists l \in [0\ldots j-1])s_l = m-1) \tag{6}$$
$$(\forall m \in [1\ldots n-1])(\exists l \in [0\ldots km-1])s_l = m \tag{7}$$

and non-isomorphic $ICDFA_\emptyset$ with n states and input alphabet Σ with k symbols.

In the canonical string representation $(s_j)_{j\in[0\ldots kn-1]}$ we can define *flags* $(f_j)_{j\in[1\ldots n-1]}$ that will be sequence of indexes of first occurrence of state label j. The initial sequence of flag is $(ki-1)_{i\in[1\ldots n-1]}$. The rules described before can be reformulated as

$$(\forall j \in [2\ldots n-1])(f_j > f_j - 1) \tag{8}$$
$$(\forall m \in [1\ldots n-1])(f_m < km) \tag{9}$$

For given k and n, the number of sequences $(f_j)_{j\in[1,n-1]}$, $F_{n,k}$ can be computed by

$$F_{k,n} = \binom{kn}{n}\frac{1}{(k-1)n+1} = C_n^{(k)}, \tag{10}$$

where $C_n^{(k)}$ are the Fuss-Catalan numbers. The proof can be found in Almeida (2009).

The process of enumeration of all possible *ICDFA* presented by canonical string representation consists from two parts: generating flags and generating all sequences inside the flag.

We define $\mathrm{cSR_S(FST)}$ as a special case of $\mathrm{SR_S(FST)}$ system, where $SR_S(FSM.transition)$ is represented by *canonical string representation*. This approach ensures that there are no isomorphisms and machines with unreachable states in the enumeration list.

4.2 Search space structure

The *normal form string* representation method allows us to define non-intersecting subspaces in the search space (Fig. 10). This gives the possibility to handle those subspaces separately in space or in time. Those subspaces are characterized by flags.

Definition 3. The universe is a set of all possible *FST*s, represented by canonical string representation $\mathrm{cSR_S(FST)}$, where canonical string representations of transition functions belong to one flag.

Definition 4. The multiverse is a set of all possible *Universes* defined by flags.

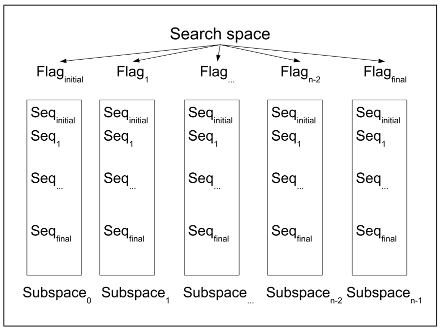


Fig. 10. Search space structure

5 Search algorithm

The task of the search algorithm is to find a point in a search space, which corresponds to the *FST* with the optimal behavior. For each point in a search space *evaluation function* assigns a score value from $[0 \dots 1]$, which describes how well corresponding *FST* behaves. So, the task is to find the point with maximal value (1.0) or at least the point with maximal score value.

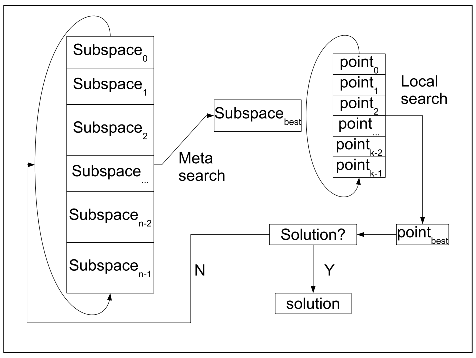


Fig. 11. Search algorithm

Search space (Multiverse) (see Definition 4) consists from several *Universes* (non-intersecting subsets) (see Definition 3) and each *Universe* contains points (see Subsection 4.2). Some of the *Universes* have higher probability to contain the solution than others. The idea is to subdivide the search algorithm into two phases, the first one chooses the *Universe* (which is 'better' than others) and second phase searches this *Universe* locally, to find the best point (Fig. 11):

1. Phase 1. 'Multiverse search'. During meta search we try to evaluate the *Universes* (subspaces), so that it is possible to choose the *Universe*, which includes solutions with higher probability.
2. Phase 2. '*Universe* local search'. The task of the local search is to find the maximal point inside pre-given *Universe*. If solution was not found the algorithm will return to the meta search phase. This cycle is done until solution (with maximal value) is found or all Universes are explored. If all subspaces are explored and solution with value 1.0 was not found, then algorithm will return the solution with maximal value (< 1.0).

5.1 Multiverse search

The task of meta search algorithm is to assign value to each subspace (described by *flag*) and to choose 'the best' subspace, which contains the solution with higher probability.

```
Size of search space: 9487698075
  0 Universe: (1, 3, 5, 7): * 1 * 2 * 3 * 4 * *  Size: 600 * 59049  From: 0 to: 599
 1 Universe: (1, 3, 5, 6): * 1 * 2 * 3 4 * * *  Size: 750 * 59049  From: 600 to: 1349
 2 Universe: (1, 3, 4, 7): * 1 * 2 3 * * 4 * *  Size: 800 * 59049  From: 1350 to: 2149
 3 Universe: (1, 3, 4, 6): * 1 * 2 3 * 4 * * *  Size: 1000 * 59049  From: 2150 to: 3149
 4 Universe: (1, 3, 4, 5): * 1 * 2 3 4 * * * *  Size: 1250 * 59049  From: 3150 to: 4399
...

 37 Universe: (0, 1, 2, 7): 1 2 3 * * * * 4 * *  Size: 6400 * 59049  From: 108150 to: 114549
 38 Universe: (0, 1, 2, 6): 1 2 3 * * * 4 * * *  Size: 8000 * 59049  From: 114550 to: 122549
 39 Universe: (0, 1, 2, 5): 1 2 3 * * 4 * * * *  Size: 10000 * 59049  From: 122550 to: 132549
 40 Universe: (0, 1, 2, 4): 1 2 3 * 4 * * * * *  Size: 12500 * 59049  From: 132550 to: 145049
 41 Universe: (0, 1, 2, 3): 1 2 3 4 * * * * * *  Size: 15625 * 59049  From: 145050 to: 160674
```

Fig. 12. The structure of the search space: subspaces and their flags

Fig. 12 illustrates the structure of the search space for the case of *Mealy machine* with 5 states. It consists of 42 *Universes* characterized by *flags*: from $(1, 3, 5, 7)$ to $(0, 1, 2, 3)$. The only useful information we know about subspace is its *flag*. *Flag* gives the pattern for sequences and basically defines how states are connected (some part of transition function). The idealistic meta search algorithm evaluates subspaces without generating points in any of them. The realistic meta search algorithm evaluates subspaces based on generated points.

The simplest idea is to evaluate the *Universe* subspace based on average value of all points. The idea of this method is to generate randomly some amount of points (defined by per cent from the size of the search space) and based on values of those points construct the subspace score. This method uses the average function to get the score. Current method uses the maximal value of the points to get the score function $Universe.score = best(Points[k].value)$ for *Universe*.

5.2 Universe local search. Gravitationally inspired search algorithm

For the *Universe* local search we propose to apply heuristic search method. We reuse the ideas presented in *PSO* algorithm (Section 3.2) and in gravitationally inspired algorithm (Section 3.3), although both of them are for continuous search space we adapt their main ideas to discrete search space. The proposed method is called 'Discrete Gravitational Swarm Optimization' (*DGSO*).

Each point in a search space characterized by:

- $position[]$ – which represents the solution,
- $velocity[]$ –which stores the information about the change of the $position[]$ vector,
- $mass$ – corresponds to score value of the solution,
- $position[]_{best}$ – the best known position for this point,
- $position[]_{global}$ – the best known position in explored search space.

First of all, we will define the distance between points. Each point, in our case, is the n-dimensional vector of integers. To calculate the distance between vectors, we use distance in one dimension

$$distanceInD(valueD_1, valueD_2) = \begin{cases} 0 : valueD_1 == valueD_2 \\ 1 : valueD_1 \neq valueD_2 \end{cases}, \tag{11}$$

which returns '1', if values in corresponding dimension are not equal and otherwise it returns '0'.

The distance between vectors is defined as sum of distances for all dimensions:

$$distance(p_1[], p_2[]) = \Sigma_{d=0}^{p.length-1} distanceInD(p_1[d], p_2[d]) \tag{12}$$

Now we need to redefine two operations: movement (change of the position) and acceleration (change of the velocity). In *PSO* and *GSA* algorithms those operations are defined as sums, in our case we will use algorithms similar to 'crossover' operator in *EA*.

Algorithm 1 $Accelerate(Force_p, Force_g, velocity[], position[]_{pBest}, position[]_{gBest})$

```
for $dimension = 0 \to velocity.length - 1$ do
  if $Random() < Force_p$ then
    $newVelocity[dimension] = position_{pBest}[dimension]$
  else
    $newVelocity[dimension] = velocity[dimension]$
  end if
end for
for $dimension = 0 \to velocity.length - 1$ do
  if $Random() < Force_g$ then
    $newVelocity[dimension] = position_{gBest}[dimension]$
  else
    $newVelocity[dimension] = velocity[dimension]$
  end if
end for
return $newVelocity[]$
```

The ’move’ operator (Algorithm 2) is the procedure, which changes the current position to the new one according to the tendency, which described by $velocity[]$ vector. The ability of changing is described by $mass_{inertial}$, the bigger $mass_{inertial}$ means, that this point tends to save it’s current position. The ’move’ operator is similar to uniform ’crossover’ operator between $position[]$ and $velocity[]$ vectors.

Algorithm 2 $Move(position[], velocity[], mass_{inertial})$

```
for $dimension = 0 \to position.length - 1$ do
  if $Random() < 1 - mass_{inertial}$ then
    $newPosition[dimension] = velocity[dimension]$
  else
    $newPosition[dimension] = position[dimension]$
  end if
end for
return $newPosition[]$
```

The proposed gravitationally inspired search algorithm was evaluated in a context of Diophantine Equation Solver by Spichakova (2016) and showed good results.

6 Simulations and analysis

Before discussing the experiments and algorithm effectiveness, we describe the complexity of the search space.

6.1 Size of the search space

Search space complexity shows how many points (which can be considered as *FST*) there are in a search space for given number of states n, size of input alphabet $k = 2$ and size of the output alphabet $m = 3$ (we leave out `W (WAIT)` action, see Subsection 1.2).

Table 1. Search space complexity

n	$cSR_S(Ant)$	$SR_S(Ant)$
2	972	1 296
3	157 464	531 441
4	34 432 128	429 981 696
5	9 487 698 075	576 650 390 625
6	3 152 263 549 140	1 156 831 381 426 180
7	1 225 311 951 419 010	3 243 919 932 521 510 000

Table 1 shows how search space grows with change of number of states n. Columns $cSR(Ant)$ and $SR(Ant)$ show the size of search space (the number of all possible *FST*) with respect to corresponding string representation: $\mathrm{cSR_S(FST)}$ and $\mathrm{SR_S(FST)}$. As you can see, proposed $\mathrm{cSR_S(FST)}$ string representation significantly reduces the size of the search space.

6.2 How to compare the effectiveness of solutions

Due to different methods and algorithms for *artificial ant* representation (grammars (Sugiura 2016), trees, *FSM*s, neural nets etc.), different heuristic search methods (*GP*, *GA*, *Ant Colony Optimization* (Chivilikhin et al. 2013)) and different possible *evaluation function* the proposed method can not be directly compared to already existing solutions. Also, we minimized and structured the search space, so it is hard to separate the effect of the proposed gravitationally inspired search algorithm from the effect of the search space minimization.

One of the most time consuming process during the search algorithm is evaluation of the ant. To optimize the search process we need to minimize the number of evaluations. *The number of fitness evaluations* ($\mathrm{N_{eval}}$) shows how many times the evaluation function was computed. $\mathrm{N_{eval}}$ is also applicable for different search methods, so it can be used as metric for algorithm comparison.

6.3 Experimental results

During experiments we have tried to construct *MeFST* with $5 \ldots 7$ states, which models the *artificial ant*. The ant is simulated on *Santa Fe Trail*. The input and output alphabets are defined as $\Sigma = \{E,\ F\}$ and $\Delta = \{M,\ L,\ R\}$, the objective function counts the number of food eaten during 400 steps using ((1)).

The parameters for the proposed *DGSO* method are the following:

- number of steps in each local search phase $e = 50$,
- at the initial stage of the *Multiverse* search some amount of point for *Universe* must be generated. Coefficients C_t (coefficient for transition function) and C_o (coefficient for output function) (Table 2) show how many points are generated:
  - $|Universe(MeFST.transition)| = |cSR_s(MeFST.transition)| \cdot C_t$
  - $|Universe(MeFST.output)| = |cSR_s(MeFST.output)| \cdot C_o$

Table 2. Initialization parameters

n	C_t	C_o
5	0.002	0.002
6	0.00015	0.00015
7	0.00005	0.00005

We use $\mathrm{N}_{\mathrm{eval}}$ (see Subsection 6.2) for measuring the quality of proposed *DGSO* method. Christensen, S. and Oppacher, F. (2007) proposed a method with $\mathrm{N}_{\mathrm{eval}}$ = 20696 fitness evaluations for *Santa Fe Trail* based on *GP* + small tree analysis. In Table 3 the third column contains the results of the method proposed by Chivilikhin et al.(2013) with respect to number of states in the *FSM*.

Table 3. Fitness evaluations

Christensen (2007)	n	*Chivilikhin (2013)*	DGSO (avg)	DGSO (min)
20696	5	*10975*	114912	**4642**
20696	6	*9313*	233508	**5205**
20696	7	*9221*	423208	93290

Table 3 contains the result for a proposed *DGSO* method: the mean number of evaluations and the minimal number of evaluations in 100 runs. The arithmetic mean of $\mathrm{N}_{\mathrm{eval}}$ for *DGSO* method is bigger than for the Chivilikhin (2013) and Christensen (2007) methods, but the minimal $\mathrm{N}_{\mathrm{eval}}$ is better for the case of 5–6 states.

6.4 Analysis

Despite the fact that the mean number of evaluations $\mathrm{N}_{\mathrm{eval}}$ is bigger than for the other existing methods, the proposed *DGSO* method has lot of potential and for some cases was able to find the solution with smaller $\mathrm{N}_{\mathrm{eval}}$. The results with smaller $\mathrm{N}_{\mathrm{eval}}$ were produced, when the method was able to find the correct *Universe* (the *Universe*, which contains the solution) during meta-search stage in the first step. The results with bigger $\mathrm{N}_{\mathrm{eval}}$ were produced during runs, where several *Universes* (25 for the case of 5 states and 34 for the case of 6 states) were searched before the solution was found.

There are two main problems with proposed method that lead to such big $\mathrm{N}_{\mathrm{eval}}$:

– During the initialization process some amount of points in each *Universe* must be generated (for test cases these parameters presented in Table 2). Currently this amount is defined with respect to size of the *Universe*. If the search space is big, as for $n = 7$, huge amount of points are generated and evaluated already at initial stage.
– The worst case scenario for the search method is the situation, when the all *Universe*s are searched and the last one contains the solution (of cause if there is solution). The best case scenario is the situation, when the solution is found in the first *Universe*. Currently, the *Universe*s are evaluated by the best point found during the initialization phase, which is not optimal.

One of the possible solutions for these two problems is to change the initialization process and modify the function for *Universe* evaluation. The ideal situation would be the possibility to evaluate the *Universe* without point generation. Such optimization can be researched in future.

7 Conclusion and future work

We presented a method for identification of *FSM*s in a context of *artificial ant* problem, which is based on heuristic optimization algorithm.

The special coding system *'Canonical String Representation'* is used to represent search space. Proposed string representation of *FSM*s is adaptation of existing method for enumeration of *FA*s, it was updated to take into account output function of *FST*. First of all, such representation allows to *minimize search space*, and, secondly, search space can be partitioned and non-intersecting partitions can be handled in *parallel* or separately in time.

The specifics of the search space representation gives the possibility to create *two stages of the search*: the search of 'best' partition (meta-search) and search inside the partition.

Due to complex structure of search space, standard *PSO* is not working well, so we propose to combine *PSO* and modern gravitational algorithm and present new hybrid *Discrete Gravitational Swarm Optimization* method. Also, the search method is modified to be able to work in discrete search space, because our search space is presented by decimal strings. So, all algorithm operators were redesigned.

The experimental results show that proposed method is efficient in some cases, but the 'meta-search phase' must be improved to make results more stable.

There are several ways to extend current research. First of all, we can adapt the proposed method to other structures, for example neural nets. Secondly, we can try the method to other similar tasks. Thirdly, the family of heuristic optimization techniques is constantly growing, so some of them can be modified to match the search space representation.

Acknowledgments

This research was supported by the Estonian Ministry of Research and Education institutional research grant no. IUT33-13.

References

Almeida, M., Moreira, M., Reis, R. (2007). Enumeration and generation with a string automata representation, *Theoretical Computer Science*, 93–102.

Almeida, M., Moreira, N., Reis, R. (2009). Aspects of enumeration and generation with a string automata representation, *CoRR*.

Angeline, P. J., Pollack, J. B. (1993). Evolutionary Module Acquisition, *Proceedings of the Second Annual Conference on Evolutionary Programming* , 154–163.

Angeline, P. J., Pollack, J. B. (1994). Coevolving High-Level Representations, *Addison-Wesley C. Langton (Eds.), Artificial Life III*, 55–71.

Bäck, T., Fogel, D.B., Michalewicz Z. (1997). Handbook of Evolutionary Computation, *IOP Publishing Ltd.*.

Chellapilla, K., Czarnecki, D. (1999). A preliminary investigation into evolving modular finite state machines *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 99*, 1349–1356 Vol. 2.

Chivilikhin, D., Ulyantsev, V., Shalyto, A. (2013). Solving Five Instances of the Artificial Ant Problem with Ant Colony Optimization *Proceedings of the 7th IFAC Conference on Manufacturing Modelling, Management, and Control*, 1043–1048.

Christensen, S., Oppacher, F. (2007). Solving the artificial ant on the santa fe trail problem in 20,696 fitness evaluations. *Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO 07*, 15741579.

Formato, R.A. (2007). Central force optimization: a new metaheuristic with applications in applied electromagnetics, *Progress in Electromagnetics Research*, 425–491.

Hsiao, Y.-T., Chuang, C.-L., Jiang J.-A., Chien C.-C.(2005). A novel optimization algorithm: space gravitational optimization, *IEEE International Conference on Systems, Man and Cybernetics*, 2323–2328, Vol. 3.

Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., Wang, A. (1991). Evolution as a theme in artificial life: The Genesys/Tracker system, *C. G. Langten, C. Taylor, J. D. Farmer, and S. Rasmussen (Eds.), Artificial life II*, 549–578.

Kennedy, J., Eberhart, R. (1995). Particle swarm optimization, *IEEE International Conference on Neural Networks*, 1942–1948, Vol.4.

Koza, J. R. (1992). Genetic programming: on the programming of computers by means of natural selection, *MIT Press*.

Kuscu, I. (1998). Evolving a Generalised Behavior: Artificial Ant Problem Revisited, *Proceedings of Seventh Annual Conference on Evolutionary Programming, LNCS*, 799–808.

Mirjalili, S., Hashim, S. Z M. (2010). A new hybrid PSOGSA algorithm for function optimization *Proceedings of International Conference on Computer and Information Application (ICCIA)*, 374–377.

Mo, S., Zeng, J., Xu, W. (2011). An Extended Particle Swarm Optimization Algorithm Based On Self-Organization Topology Driven By Fitness, *Journal of Computational Information Systems, 7:12*, 4441–4454.

Qi, K., Lei, W., Qidi, W. (2007). A Novel Self-organizing Particle Swarm Optimization based on Gravitation Field Model *American Control Conference, 2007. ACC '07*, 528–533.

Rashedi, E., Nezamabadi-pour, H., Saryazdi, S. (2009). GSA: A Gravitational Search Algorithm, *Inform. Sciences, Nr.13, Vol. 179*, 2232–2248.

Reis, R., Moreira, N., Almeida, M. (2005). On the representation of finite automata, *Proc. of DCFS05*, 269–276.

Spichakova, M. (2016). Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions, *Proceedings of the Estonian Academy of Sciences, Vol. 65, Issue 1*, 15–27.

Sugiura, H., Mizuno, T., Kita, E. (2012). Santa Fe Trail Problem Solution Using Grammatical Evolution , *2012 International Conference on Industrial and Intelligent Information (ICIII 2012)*, 36–40.

Tsai, H.-C., Tyan, Y.-Y., Wu, Y.-W., Lin, Y.-H.(2013). Gravitational Particle Swarm, *Applied Mathematics and Computation, Vol. 219, Num. 17* , 9106–9117.

Webster, B., Bernhard, P. J. (2003). A Local search optimization algorithm based on natural principles of gravitation, *CS-2003-10, Florida Institute of Technology*.

Webster, B. (2004). Solving combinatorial optimization problems using a new algorithm based on gravitational attraction, *Ph.D. thesis, Florida Institute of Technology*.

Received September 9, 2016 , revised December 12, 2016, accepted February 21, 2017

Publication B

Spichakova, Margarita (2016). ***Modified Particle Swarm Optimization Algorithm Based on Gravitational Field Interactions.*** *Proceedings of the Estonian Academy of Sciences, 65(1), 15 – 27.*

Proceedings of the Estonian Academy of Sciences,
2016, **65**, 1, 15–27
doi: 10.3176/proc.2016.1.01
Available online at www.eap.ee/proceedings

COMPUTER
SCIENCE

Modified particle swarm optimization algorithm based on gravitational field interactions

Margarita Spichakova

Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia; margo@cs.ioc.ee

Received 9 February 2015, revised 11 May 2015, accepted 25 May 2015, available online 10 December 2015

Abstract. In this paper we present the modified particle swarm optimization algorithm, where gravitational interactions between particles are used for computing learning coefficients. The behaviour of the algorithm is demonstrated by solving the two-dimensional Diophantine equation problem. This allows us to observe the search space and workflow of the algorithm directly on the two-dimensional plane.

Key words: particle swarm optimization, Diophantine equation solver, gravitationally inspired heuristic search.

1. INTRODUCTION

Define the *search problem* as finding an optimum (minimum or maximum) of some given function. The set of *points*, presenting the function arguments, gives us the *search space*. For each point in the search space there is the value of the function. The task is to find the point (argument), which gives the optimal value of the function.

If no information about the search space is available, two main opportunities exist: (1) to exhaustively traverse the search space or (2) to choose some random points and pick up the most suitable one. If the search space is infinite, we define some *range*. Traversing the whole search space takes time, but gives an exact result. In the case of an infinite search space, the results also depend on the definition of the range, and it may happen that the range does not contain the optimum. Randomly generating a small number of points in the search space gives a solution very fast, but the quality of the solution is questionable. The stochastic optimization presents the compromise between exhaustive search and choosing random points.

Optimization techniques include several approaches, some of which are *deterministic* procedures and others contain randomness and probabilistic computations. The main advantage of *stochastic optimization* is that it can be applied to any search problem without specific knowledge about the structure of the search space. Stochastic optimization may also be helpful when the complexity of deterministic methods grows rapidly with the search space size.

Two main properties must be implemented in any stochastic optimization method: the *exploration* and the *exploitation*. The exploration is the ability of the method to explore the entire search space in a global way and the exploitation is its ability to focus in the local area and search for a more exact solution.

Stochastic optimization methods have several ideas in common:

- The *search space* is defined as a set of points where each point represents a *candidate solution*. Usually, candidate solutions are presented indirectly, by some structure, which encodes the candidate solution. Initially, some fixed number of points are generated randomly.

- By using the *evaluation function* we can assign the *score value*, which shows how good a solution is.
- The search algorithm contains *modification operators* that allow the construction of new solutions from the existing ones.

The nature has been the source of inspiration for constructing new stochastic search algorithms. There is a set of methods, such as evolutionary algorithms, genetic algorithms, evolutionary programming, which are based on the theory of evolution. Some methods simulate social behaviour, for example, *particle swarm optimization (PSO)* imitates the social behaviour of birds, ant colony optimization applies ideas of the behaviour of ants foraging for food. Some stochastic optimization methods are based on the laws of physics, for example, simulated annealing is based on the thermodynamic effect. Others are based on gravitational force. For example, *Central Force Optimization* is a deterministic gravity-based search algorithm, which simulates the group of probes [2]. *Space Gravitational Optimization* [3] simulates asteroids flying through a curved search space. A gravitationally-inspired variation of local search, the *Gravitational Emulation Local Search Algorithm*, was proposed by Webster [14] and Webster and Bernhard [15]. The newest one, the *Gravitational Search Algorithm*, was proposed by Rashedi et al. [10–12] as a stochastic variation of Central Force Optimization. A discrete modification of the Gravitational Search Algorithm was proposed by Zibanezhad et al. [16] in the context of Web-Service composition.

In this article we discuss the family of PSO algorithms. Standard PSO contains three parameters to control the algorithm. Traditionally, the values of the parameters are defined by end users. In general, no exact methods exist for defining the parameters, so, usually, it is done empirically. Also, it is possible to define a new search problem for finding values of parameters and applying stochastic optimization. This process is called *meta-heuristics*.

We present a modification of the PSO algorithm based on gravitational interactions (GI) between particles (PSO + GI), which automatically adjusts the parameters of the PSO algorithm. The proposed method solves the parameter adjusting problem by replacing parameters with computed values.

Both methods, the standard PSO algorithm and the proposed modification PSO + GI algorithm, are tested on the Diophantine equation solver task (see Table 1 in Section 5.1 for test equations). This search problem is chosen for illustrative purposes.

The paper is organized as follows. Section 2 defines the search problem, Section 3 presents the standard PSO algorithm, and Section 4 describes the new method PSO + GI. Section 5 covers the behaviour of the proposed method and Section 6 contains the conclusion and future plans.

2. DIOPHANTINE EQUATION SOLVER

A *Diophantine equation (DE)* has the form

$$F(x_1, x_2, \ldots, x_m) = 0, \tag{1}$$

where coefficients and variables are integers and F can be considered as a polynomial function.

For simple cases there are deterministic methods for solving such equations. For example, in the case of a linear DE with two variables, the solution can be found by using the Euclidean algorithm for the greatest common divider.

The problem of finding a general deterministic method for solving any DE is known as 'Hilbert's tenth problem'. It was proven by Y. Matiyasevich in 1970 that there is no such deterministic method. Therefore, the use of stochastic optimization methods is helpful.

Several stochastic optimization algorithms are used for solving DEs. Abraham [1] applied the standard PSO algorithm to the DEs of simpler forms (see Eq. 2) and tested it on two sets: the first set – DEs with n from 2 to 15 (see Eq. 2) and the second one with equations with power 2 with 2 to 12 variables. In all cases the PSO method was able to find the solutions.

The genetic algorithm can also be used for solving a DE. For example, there is a genetic algorithm tutorial [4], where the genetic algorithm is demonstrated on solving the equation $a + 2b + 3c + 4d = 30$.

To use the stochastic optimization method for the DE solver problem, we need to define at least three things: the search problem, search space, and evaluation function.

The dimensionality of the search space depends on the number of arguments to be found. The restriction $m = 2$ gives us only two variables x and y, so the general DE has the form

$$a \cdot x^n + b \cdot y^n = d. \tag{2}$$

The test equations are presented in Table 1 in Section 5.1.

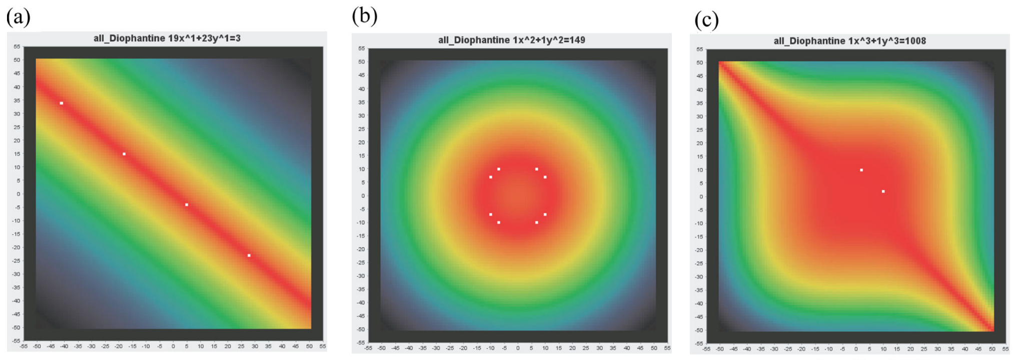
The *evaluation function* must return the value for each point (candidate solution) in the search space. In our case, the candidate solution is a pair (x_p, y_p). In the easiest case, we can try a point of the search space (x_p, y_p) as a solution of the DE and get the result *true* or *false*. Unfortunately, this gives not much information about how close this point was to optima.

We can define the evaluation function as a distance between two points:

$$\begin{aligned} a \cdot x^n + b \cdot y^n &= d, \\ a \cdot x_p^n + b \cdot y_p^n &= dd, \\ f &= |dd - d|. \end{aligned} \tag{3}$$

If we apply our candidate solution (x_p, y_p) to the equation, we can compute dd. In our original Eq. 2, this value is equal to d. If $dd = d$, then (x_p, y_p) is our optimal solution. To measure how far the candidate solution is from the unknown optimum, we define the distance f, which will be our evaluation function. So, the search problem is now defined as *minimize the evaluation function f*. The point (x_p, y_p) will be considered as an optimal solution if $f = 0$.

As we mentioned before, the search space is defined as a set of candidate solutions (x_p, y_p). We also restricted the search area by upper and lower bounds, which are defined by the user. The coordinates (x_p, y_p) present two dimensions and the evaluation function value gives the third one. To illustrate such a search space, we will use the diagram where points with coordinates correspond to candidate solutions and the evaluation function value is coded by colour in such a way that the colour ranges from blue (that corresponds to long distances) to red (that corresponds to smaller distances), and the optimal values are presented by white colour. Examples of such diagrams are shown in Fig. 1.



$19x + 23y = 3$ $x^2 + y^2 = 149$ $x^3 + y^3 = 1008$

Fig. 1. Examples of the search space.

3. PARTICLE SWARM OPTIMIZATION

The PSO algorithm is inspired by social behaviour of some set of objects, for example, bird flock or fish school [5]. The general idea of this method can be described as follows. The search space contains a set of points and each point has its value, which is assigned by the *evaluation function*. The *task* is to find the optima of this function. There is also a set of particles that are moving on the defined search space. The movement laws can be considered as interactions between objects. Using those movement laws, the objects must find the positions with better values or even optima.

3.1. Standard PSO algorithm

Consider a set of particles – a *swarm*. Each particle is characterized by the *position vector*, *velocity vector*, and the best known position for this object. There exists also the global best known position for the whole swarm.

The *position* vector $p_d, d \in [0 \dots n]$ presents a *candidate solution*. The dimensionality n of the vector depends on the problem size. We assign the value for each candidate solution using the evaluation function. According to those values we can choose the global best known position *Gbest*, which is the point with the optimal value found so far by the whole swarm, and the local best known position *Pbest*, which is the best position that was found by this exact particle.

The *velocity* vector $v_d, d \in [0 \dots n]$ represents the trend of movement of the particle. It is computed by using the equation

$$v_d(t) = \alpha \cdot v_d(t-1) + \beta \cdot r_1 \cdot (Pbest_d - p_d(t-1)) + \gamma \cdot r_2 \cdot (Gbest_d - p_d(t-1)), \tag{4}$$

where

- α, β, γ are learning coefficients with α representing the inertia, β is the cognitive memory, and γ is the social memory; those coefficients must be defined by the user;
- r_1 and r_2 are random values in the range $[0 \dots 1]$;
- $Pbest_d$ is the local best known position for the particle and $Gbest_d$ is the global best known position of the swarm;
- $v_d(t)$ is the new value of the velocity vector at dimension d and $v_d(t-1)$ is the previous value of the velocity.

The new position $p_d(t)$ is simply defined as the sum of the previous position $p_d(t-1)$ and the new velocity $v_d(t)$:

$$p_d(t) = p_d(t-1) + v_d(t). \tag{5}$$

The PSO algorithm is presented in Algorithm 1. The initialization part consists of defining the required learning parameters, setting up boundaries of the search space, and generating the swarm with a random position and velocity. The search process is an iterative update of the positions and velocities. The process ends when the ending criteria are met: either the number of iterations is exceeded or the optimal solution is found. The evaluation function f and the dimensionality of vectors are problem-specific.

3.2. The behaviour of the standard PSO algorithm

Algorithm 1 contains three learning coefficients α, β, γ for adjusting the convergence abilities of the algorithm. Learning coefficients must be defined by the user according to the problem statement. No deterministic methods are available for finding their values. However, there are several non-deterministic methods for solving this problem. For example, in the case of the empirical methods we can try several parameter values and observe the behaviour of the PSO algorithm and choose the best ones. In the case of meta-heuristics, the choice of the parameter values can also be considered as a search problem, so here

Algorithm 1. Standard particle swarm optimization

```
Set bounds for the search space $B_{up}$, $B_{low}$
Set learning coefficients $\alpha$, $\beta$, $\gamma$
Define the size of the swarm $s$ and the number of iterations $e$
for $i = 0 \to s-1$ do
  Initialize particle position $p^i$ taking $B_{up}$ and $B_{low}$ into account
  $Pbest^i \leftarrow p^i$
  Initialize particle velocity $v^i$ taking $B_{up}$ and $B_{low}$ into account
  Evaluate particle $f(p^i)$
  if $f(Gbest^i) < f(p^i)$ then
    $Gbest^i \leftarrow p^i$
  end if
end for
while current iteration $< e$ and optimal solution is not found do
  for $i = 0 \to s-1$ do
    Update velocity $v^i$ for each dimension by Eq. 4
    Update position $p^i$ for each dimension by Eq. 5
    if $f(Pbest^i) < f(p^i)$ then
      $Pbest^i \leftarrow p^i$
      if $f(Gbest^i) < f(p^i)$ then
        $Gbest^i \leftarrow p^i$
      end if
    end if
  end for
end while
```

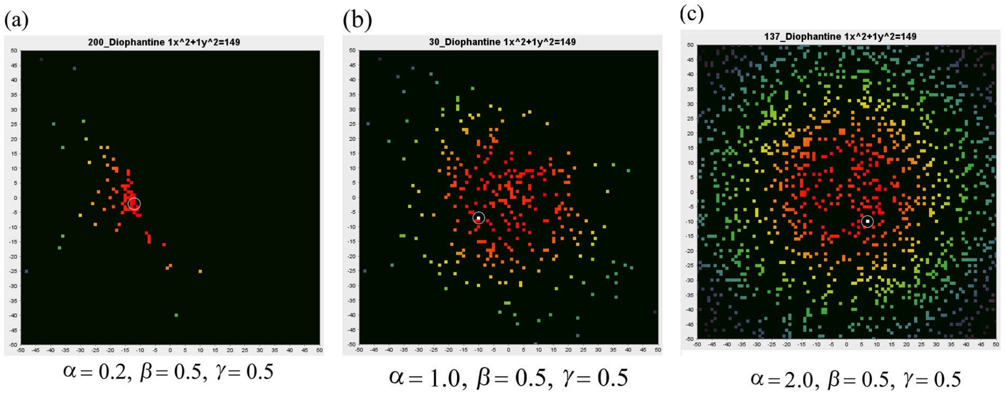


Fig. 2. Equation: $x^2 + y^2 = 149$. Maximum number of iterations $e = 200$.

heuristic optimization can also be applied. The unknown values of the learning coefficients constitute one of the problems with the PSO algorithm.

Example 3.1. Figure 2 shows how different values of a learning coefficient can affect the optimization process. We use the equation $x^2 + y^2 = 149$ and the same initial swarm which was generated randomly. The maximum number of iterations is 200 for all three cases. The cognitive memory $\beta = 0.5$ and social memory $\gamma = 0.5$ stay the same, only inertia α is changing:

- Figure 2a shows the case where $\alpha = 0.2$. Such a small α value leads to fast convergence of the search process to the global best value *Gbest*. The exploration ability of PSO in this case is small.
- Figure 2b shows the case where $\alpha = 1.0$. The solution was found on 30 iterations. With these parameters the algorithm was able to find the optimal solution in most cases. The search process is going on inside

the ‘red zone’. It means that first of all the algorithm was able to define where there is a good zone for searching (exploration) and then explore this zone closely (convergence).

- Figure 2c shows the case where $\alpha = 2.0$. In this case the optimal solution was found during 137 iterations. However, the algorithm was exploring the whole search space, so much additional work was done. We can say that in this case the algorithm does not converge to the optimal solution.

3.3. Problems with the standard PSO algorithm

As we have shown in Example 3.1, the choice of the learning coefficients for the PSO algorithm has a strong impact on optimization performance. The first problem is that no exact methods exist for defining them.

The second problem lies in the definition of Eq. 4. There is the *Gbest* position, which is valid for the whole swarm and does not take the distance between the particle and the global best position *Gbest* into account. In some cases, if *Gbest* itself is in a bad zone, the whole swarm falls into a local optima.

The second problem has two main solutions: (1) to define the *neighbourhood* of every particle by taking account of not *Gbest* for the whole swarm, but of *Gbest* for the group of ‘connected’ particles, (2) to define *parallel swarms*, where groups of particles move in the search space without any interactions between groups.

4. PROPOSED ALGORITHM: MODIFIED PSO BASED ON GRAVITATIONAL FIELD INTERACTIONS

To solve the problems described above, we propose not to define learning coefficients by the user, but to compute them in such a way that they take the distances between particles into account. We employ the ideas inspired from stochastic search methods based on the gravitational law.

4.1. Gravity as inspiration for optimization algorithms

Four main forces are acting in our universe: gravitational, electromagnetic, weak nuclear, and strong nuclear. These forces define the way our universe behaves and appears. The weakest force is gravitational; it defines how objects move depending on their masses.

The gravitational force between two objects *i* and *j* is directly proportional to the product of their masses and inversely proportional to the square distance between them

$$F_{ij} = G\frac{M_j \cdot M_i}{R_{ij}^2}. \tag{6}$$

Knowing the force acting on the body, we can compute acceleration as

$$a_i = \frac{F_i}{M_i}. \tag{7}$$

To construct the search algorithm based on gravity, we can use the following ideas:

- each object in the universe has mass and position;
- there are interactions between objects, which can be described by using the law of gravity;
- bigger objects (with greater mass) create a larger gravitational field and attract smaller ones.

During the last decade some researchers have tried to adapt the idea of gravity to find out optimal search algorithms. Such algorithms have some general ideas in common:

- the system is modelled by objects with mass;

- the position of the objects describes the solution and the mass of the objects depends on the evaluation function;
- the objects interact with each other using gravitational force;
- the objects with greater mass present the points in the search space with better solution.

Using these characteristics, it is possible to define the family of optimization algorithms based on gravitational force. For example, *Central Force Optimization* is a deterministic gravity-based search algorithm proposed and developed by Formato [2]. It simulates the group of probes which fly into the search space and explore it. Another algorithm, *Space Gravitational Optimization*, was developed by Hsiao et al. [3] in 2005. It simulates asteroids flying through a curved search space. A gravitationally-inspired variation of the local search algorithm, *Gravitational Emulation Local Search Algorithm*, was proposed by Webster [14] and Webster and Bernhard [15]. The newest one, the *Gravitational Search Algorithm*, was introduced by Rashedi et al. [11] as a stochastic variation of Central Force Optimization.

Basically, the gravitationally inspired algorithms are quite similar to PSO algorithms. Instead of the particle swarm we have a set of bodies with masses, ideas of the position and velocity vectors are the same, and the movement laws are similar. Our idea is to combine the two approaches to get a better one.

4.2. Existing PSO algorithm hybrids

The idea of hybridization of the PSO algorithm and gravitationally inspired search algorithms is not new. Several algorithms exist that use both ideas (PSO algorithm and gravity) to construct a heuristic search algorithm:

- PSOGSA – PSO algorithm and Gravitational Search Algorithm [7];
- extended PSO algorithm based on self-organization topology driven by fitness – PSO algorithm and Artificial Physics [8];
- Gravitational Particle Swarm [13];
- self-organizing PSO based on the Gravitation Field Model [9].

The traditional way of hybridization of the PSO algorithm with gravitationally-inspired search algorithms is to add the gravitational component to the velocity computation. Equation 4 has an additional component, which is computed by using gravitational interactions. Unfortunately, this makes the behaviour of the search algorithm even more complex and unpredictable. Additionally, the user-defined parameters still need to be found.

We propose not to add the gravitational component, but to replace the existing learning coefficients by coefficients which are computed by 'gravitational' interactions.

4.3. Modified PSO based on gravitational field interactions

Now we want to adapt some ideas we got from the gravitational search to the PSO algorithm. Suppose we have the current particle P, its position and velocity vectors, and its mass M, which is based on the value of the evaluation function. The position vector encodes the candidate solution, and the velocity vector presents the direction of the movement. We know the *Gbest* position and mass value M_g for this position. As for the standard PSO algorithm, so far this *Gbest* position presents the best known solution for the whole swarm. We also have the local best value for the current particle – the *Pbest* position and its mass value M_b at this point.

For now, the algorithm is similar to the standard PSO. The current particle is attracted to both *Pbest* and *Gbest* positions; in the PSO algorithm the power of this attraction and direction of the movement is controlled by learning coefficients. In our case, we recalculate the force between those three points and compute the acceleration (Fig. 3).

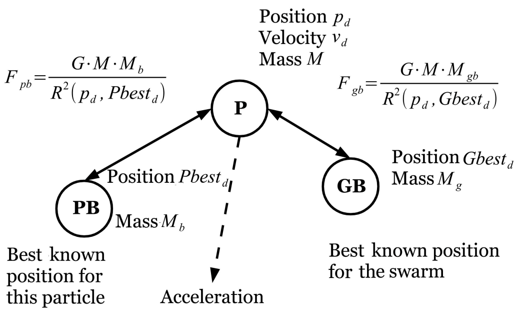

Position p_d
Velocity v_d
Mass M
$F_{pb} = \frac{G \cdot M \cdot M_b}{R^2(p_d, Pbest_d)}$
$F_{gb} = \frac{G \cdot M \cdot M_{gb}}{R^2(p_d, Gbest_d)}$
P
PB
GB
Position $Pbest_d$
Mass M_b
Position $Gbest_d$
Mass M_g
Best known position for this particle
Acceleration
Best known position for the swarm

Fig. 3. Gravitational interactions between particles.

In the proposed method we replace learning coefficients in the standard velocity computation by computed values, which are based on the idea of gravitational interactions between particles:

$$v_d(t) = \mathbf{M_i} \cdot v_d(t-1) + \mathbf{a_{pb}} \cdot r_1 \cdot (Pbest_d - p_d(t-1)) + \mathbf{a_{gb}} \cdot r_1 \cdot (Gbest_d - p_d(t-1)), \tag{8}$$

where

- M_i (instead of α) – inertial mass,
- a_{pb} (instead of β) – acceleration towards *Pbest*

$$a_{pb} = \frac{G \cdot M_b}{R^2(p(t-1), Pbest)}, \tag{9}$$

- a_{gb} (instead of γ) – acceleration towards *Gbest*

$$a_{gb} = \frac{G \cdot M_g}{R^2(p(t-1), Gbest)}, \tag{10}$$

- r_1,r_2, $Pbest_d$, $Gbest_d$, $v_d(t)$, d, $v_d(t-1)$ have the same meaning as in Eq. 4.

Now we have another movement law. Learning coefficients β and γ are replaced by the corresponding accelerations, based on gravitational interaction between the corresponding particles. The ability of the particle to save its current position is now characterized by M_i instead of α.

Contrary to the standard PSO algorithm, where learning coefficients are chosen by the end user, in our case learning coefficients are computed by using gravitational interactions between particles. Moreover, the proposed learning coefficients depend on distances between the position of the current particle *P* and *Pbest* position and the position of *P* and *Gbest* position. Thus, we have an additional property: if the current particle is far from *Gbest*, the tendency to move in that direction is small. Therefore, the behaviour of the algorithm is more stable than that of the standard PSO.

5. BEHAVIOUR OF THE PROPOSED PSO + GI METHOD

To illustrate the PSO + GI method, we perform several experiments and compare the results with other similar algorithms, such as PSO and *stochastic hill climbing (SHC)* [6].

5.1. Experimental setup

To analyse the behaviour of the proposed algorithm, we use several test problems that can be described by Eq. 2 with powers from 1 to 5. The problems are presented in Table 1. The search space is defined in the

Table 1. Test Diophantine equations

	Equation	Solutions in the range
e1	$4x+9y=91$	11
e2	$10x+7y=97$	10
e3	$24x+15y=9$	13
e4	$19x+23y=3$	4
e5	$x^2+y^2=625$	20
e6	$x^2+y^2=149$	8
e7	$x^3+y^3=1008$	2
e8	$x^4+y^4=1921$	8
e9	$x^5+y^5=19\,932$	2

range $[-50,50]$ for each variable. The column 'Solutions in the range' shows how many solutions exist in the search space.

The initial swarm is generated in the range $[-50,50]$ for each variable and its size is defined as 1% of all possible points in the search space. The exception is the experiment 'Bad initial set' (see Subsection 5.3). For this experiment the inital swarm is generated in the range $[-50,-45]$ for each variable, although the search space stays the same ($[-50,50]$ for each variable).

The maximum number of iterations is taken 200. So, the algorithm stops running after 200 iterations or if the solution is found.

Two main characteristics are used for comparison: space and time. In the proposed experiments, the space is described by the number of points the algorithm checked before finding the solution and time is described by the number of iterations before the solution was reached. The number of fails (the solution was not found during 200 iterations) is an important factor as well.

For each experiment we perform 1000 runs and compute average and standard deviation for each parameter used for describing algorithm performance (see Tables 2–4).

The PSO + GI algorithm is compared to SHC and standard PSO algorithm with $\alpha = 1.0$, $\beta = 0.5$, $\gamma = 0.5$ (Section 3).

5.2. Experiments: normal workflow

Search includes two main mechanisms: global and local. The performance of each search method depends on those mechanisms. One idea of the experiments was to take a look inside and observe what exactly is going on during the search process. Usually, the search problems are multidimensional and it is hard to illustrate the search space. The use of two-dimensional DEs as test problems allows us to illustrate the behaviour of the search algorithm.

Figure 4 shows one test run of the algorithm: PSO + GI in Fig. 4a,b, SHC in Fig. 4c, and PSO in Fig. 4d for test equation e1 (Table 1).

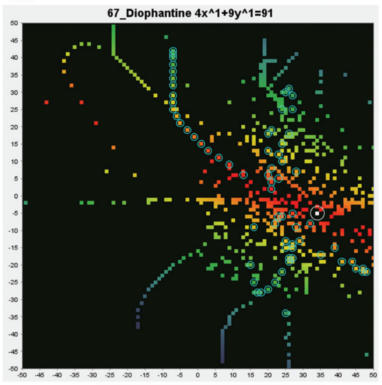
Figure 4c illustrates the typical behaviour of SHC. There are several 'islands' that are formed by moving particles. The movements of one particle also exactly represent 'hill climbing'.

Figure 4d shows the behaviour of PSO. It is hard to define any observable pattern in the behaviour.

For PSO + GI (Fig. 4a,b) we define two main trajectories of the particles: the 'line' (Fig. 4b) and the 'orbit' (Fig. 4a). The first one appears, when *Gbest* is far from the current particle position, or constantly changing. In this case the particle moves directly to better zones. The second one, the 'orbit', appears when the particle is near *Gbest*. In this case the particle starts to move around *Gbest* in the good area. These trajectories can be explained by principles of local and global search.

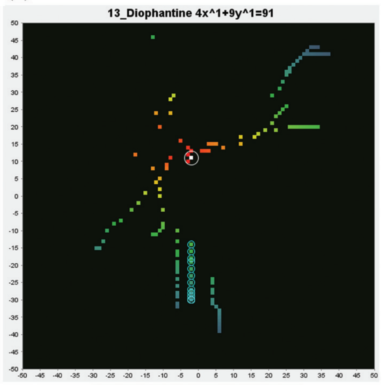
Table 2 presents the results of the experiment 'Different initial set'. We performed 1000 runs for each algorithm and each test equation. A new initial set was used (generated randomly for each run). The standard PSO showed better results than the others. The PSO + GI algorithm is best for e2 and second-best for e1, e3, e5, e6, e7, and e8. Though, PSO + GI performed almost on the same level as PSO.

(a)



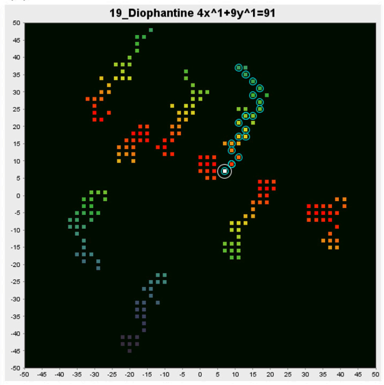
PSO + GI. Pattern: orbits

(b)



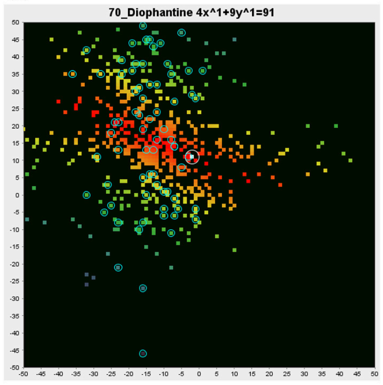
PSO + GI. Pattern: line

(c)



SHC

(d)



PSO

Fig. 4. Trajectories.

Table 2. Experiment I ‘Different initial set’

Eq.	SHC					PSO					PSO + GI				
	Iterations		Points		Fails	Iterations		Points		Fails	Iterations		Points		Fails
	Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.	
e1	73.12	84.76	231.15	110.88	305	39.16	37.94	342.22	297.07	5	49.57	44.47	431.44	356.03	14
e2	39.06	62.79	191.93	110.13	128	49.63	49.83	412.61	366.85	30	57.22	47.72	496.78	387.39	18
e3	58.63	78.05	210.89	109.93	230	38.9	40.65	335.68	309.52	8	48.46	42.87	421.51	345.87	14
e4	66.75	80.08	247.54	112.23	261	106.08	73.20	816.15	518.88	252	111.94	70.46	919.22	541.96	270
e5	21.19	44.90	135.22	72.17	58	24.86	24.50	231.54	205.57	0	28.04	27.72	260.52	242.67	1
e6	18.69	24.29	193.67	93.84	15	34.13	36.24	307.05	291.15	6	43.04	41.48	389.31	349.68	11
e7	84.97	79.07	326.93	104.10	302	76.43	65.46	641.62	506.38	115	99.60	67.96	867.40	556.39	195
e8	19.88	22.85	214.76	101.80	13	22.33	22.68	214.41	197.49	0	32.53	31.53	305.80	277.99	3
e9	51.14	55.21	325.12	116.63	95	60.23	55.26	519.82	430.704	62	87.77	66.65	778.15	558.69	151

The behaviour of the algorithm depends not only on the mechanisms of the search, but also on the initial set. To eliminate this difference, we performed a second experiment, where we used the same initial set

Table 3. Experiment II ‘Same initial set’

Eq.	SHC					PSO					PSO + GI				
	Iterations		Points		Fails	Iterations		Points		Fails	Iterations		Points		Fails
	Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.	
e1	33.46	57.33	350.40	175.69	103	17.51	18.68	322.07	292.80	0	25.45	23.63	457.38	380.24	0
e3	13.40	33.31	205.72	153.71	29	18.97	20.13	343.78	306.74	0	23.95	23.37	425.16	375.48	0
e4	11.94	23.44	248.36	134.65	14	61.91	56.11	914.85	700.88	56	71.16	60.64	1112.82	824.05	73
e5	5.02	9.71	138.05	96.80	2	12.05	12.14	238.58	206.43	0	13.47	12.77	261.55	227.90	0
e6	3.89	1.82	125.49	44.57	0	17.16	17.32	312.60	271.15	0	22.00	19.78	411.07	334.71	0
e7	45.58	36.38	566.18	168.15	36	37.97	38.29	620.27	539.56	5	61.29	52.52	1028.17	780.42	41
e8	7.47	4.13	207.71	94.50	0	9.62	10.09	196.80	174.35	0	14.74	13.73	289.16	242.68	0
e9	9.48	5.41	253.22	114.84	0	28.37	25.69	502.13	384.52	1	53.67	46.58	929.41	718.50	22

for all 1000 runs and all three algorithms. This allows us to compare pure behaviours: how the different algorithms perform on the same initial set.

Table 3 shows the results of experiment II ‘Same initial set’. The standard PSO shows the best performance, except for the problems e4 and e9. The PSO + GI algorithm is on the second place.

5.3. Experiment: bad initial set

The results of the search depend not only on the algorithm, but also on the initial swarm. What will happen if the initial swarm is generated in the bad zone of the search space? Figure 5 shows this situation for the test equation e1. The initial swarm was generated in the range $[-50, -45]$ for each variable. For test equations, this is the area with points that have bigger evaluation function values. The PSO + GI algorithm was able to move out of the bad zone towards the better one and find the solution. However, this process takes many iterations.

All the three algorithms were able to leave the bad zone. Table 4 contains the results of the third experiment ‘Bad initial sets’. As you can see, the standard PSO performs better in most cases (except e4, e6, e7, e9; for those test problems SHC outperformed the others). The PSO + GI is the second-best algorithm for the test problem: e3 and e8.

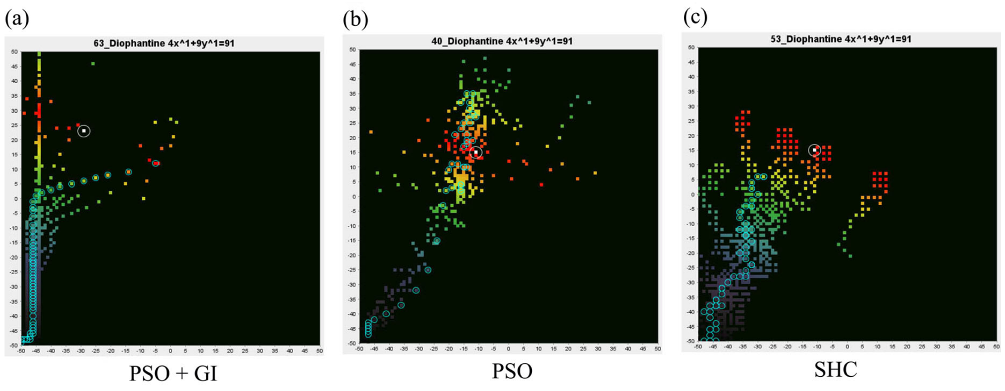


Fig. 5. Bad initial set.

Table 4. Experiment III 'Bad initial sets'

Eq.	SHC					PSO					PSO + GI				
	Iterations		Points		Fails	Iterations		Points		Fails	Iterations		Points		Fails
	Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.		Avg.	Std. dev.	Avg.	Std. dev.	
e1	93.19	68.84	551.50	86.60	291	58.48	42.78	453.64	293.77	20	132.03	61.20	478.78	303.15	321
e2	64.42	45.17	534.92	101.38	95	70.88	50.51	544.82	349.12	52	146.65	45.55	709.76	300.31	246
e3	82.53	67.16	490.06	96.84	243	55.16	39.44	435.09	283.40	8	122.11	48.08	628.06	293.04	116
e4	86.74	68.24	512.94	106.10	262	125.39	125.39	125.39	457.02	314	164.65	47.10	945.43	354.13	515
e5	56.47	59.50	285.16	39.71	146	30.04	21.13	239.60	157.98	1	47.50	32.20	386.14	263.09	5
e6	39.53	6.42	342.87	40.55	1	36.32	29.06	294.22	214.68	6	62.25	38.81	495.14	317.34	24
e7	80.30	44.64	538.26	56.11	120	95.12	64.77	753.90	483.48	170	121.27	64.39	998.90	514.44	269
e8	45.41	17.64	374.82	44.29	12	26.26	15.46	222.10	127.88	0	44.22	34.33	400.84	299.04	7
e9	68.19	30.04	512.26	54.30	47	67.31	54.46	547.99	417.80	73	100.55	64.92	884.48	540.06	178

6. CONCLUSION AND FUTURE WORK

We proposed a modified PSO algorithm PSO + GI based on ideas of gravitational interactions between bodies. This algorithm replaces predefined learning coefficients by new ones that are calculated by means of the evaluation function and distance between particles. The general idea of the algorithm is to solve the problem with unknown learning coefficients for the standard PSO. Also, in PSO + GI, the distance factor helps to resolve the problem of the global behaviour for PSO.

The PSO + GI algorithm was tested on several Diophantine equations and the results were compared to the standard PSO and SHC. The statistics show that PSO + GI shows the second-best performance after well-tuned PSO in most cases. So the main goal of PSO + GI (to reduce the number of unknown coefficients in PSO) was achieved.

In the future, the PSO + GI algorithm should be tested for other problems with more complex search spaces.

ACKNOWLEDGEMENTS

This research was supported by Estonian Research Council institutional research grant No. IUT33-13, Estonian Doctoral School in ICT (IKTDK), and by the ERDF through the ITC project MBJSDT and Estonian national CoE project EXCS. The author is grateful to the anonymous reviewers who made constructive suggestions and remarks on the draft of this paper.

REFERENCES

1. Abraham, S., Sanya, S., and Sanglikar, M. A. Particle swarm optimization based diophantine equation solver. *CoRR*, abs/1003.2724, 2010.
2. Formato, R. A. Central force optimization: a new metaheuristic with applications in applied electromagnetics. *PIER*, 2007, **77**, 425–491.
3. Hsiao, Y.-T., Chuang, C.-L., Jiang, J.-A., and Chien, C.-C. A novel optimization algorithm: space gravitational optimization. In *Systems, Man and Cybernetics, 2005 IEEE International Conference*, Vol. 3. 2005, 2323–2328.
4. Hsiung, S. and Mattews, J. Genetic algorithm example: Diophantine equation, 1999. www.generation5.org [accessed 23 May 2015].
5. Kennedy, J. and Eberhart, R. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, Vol. 4. 1995, 1942–1948.
6. Luke, S. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/ [accessed 23 May 2015].
7. Mirjalili, S. and Hashim, S. Z. M. A new hybrid PSOGSA algorithm for function optimization. In *Proceedings of International Conference on Computer and Information Application (ICCIA)*. 2010, 374–377.

8. Mo, S., Zeng, J., and Xu, W. An extended particle swarm optimization algorithm based on self-organization topology driven by fitness. *J. Comput. Inform. Syst.*, 2011, **7**(12), 4441–4454.
9. Qi, K., Lei, W., and Qidi, W. A novel self-organizing particle swarm optimization based on gravitation field model. In *American Control Conference, 2007, ACC '07*, 2007, 528–533.
10. Rashedi, E., Nezamabadi-pour, H., Saryazdi, S., and Farsangi, M. M. Allocation of static var compensator using gravitational search algorithm. In *First Joint Congress on Fuzzy and Intelligent Systems Ferdowsi University of Mashhad, Iran, August, 29–31*. 2007, 29–31.
11. Rashedi, E., Nezamabadi-pour, H., and Saryazdi, S. GSA: a gravitational search algorithm. *Inform. Sci.*, 2009, **179**(13), 2232–2248.
12. Rashedi, E., Nezamabadi-pour, H., and Saryazdi, S. Filter modeling using gravitational search algorithm. *Eng. Appl. Artif. Intell.*, 2011, **24**, 117–122.
13. Tsai, H.-C., Tyan, Y.-Y., Wu, Y.-W., and Lin, Y.-H. Gravitational particle swarm. *Appl. Math. Comput.*, 2013, **219**(17), 9106–9117.
14. Webster, B. *Solving Combinatorial Optimization Problems Using a New Algorithm Based on Gravitational Attraction*. PhD thesis, Florida Institute of Technology, Melbourne, FL, USA, 2004.
15. Webster, B. and Bernhard, P. J. *A Local Search Optimization Algorithm Based on Natural Principles of Gravitation*. Technical Report CS-2003-10, Florida Institute of Technology, 2003.
16. Zibanezhad, B., Zamanifar, K., Nematbakhsh, N., and Mardukhi, F. An approach for web services composition based on QoS and gravitational search algorithm. In *Proceedings of the 6th International Conference on Innovations in Information Technology, IIT'09*. IEEE Press, Piscataway, NJ, USA, 2009, 121–125.

Gravitatsioonilist vastasmõju arvestav osakeste parvega optimeerimise meetod

Margarita Spichakova

On esitatud osakeste parve optimeerimisalgoritmi modifikatsioon PSO + GI, kus otsimismeetodi parameetrid arvutatakse osakestevahelise gravitatsioonilise vastasmõju põhjal. PSO + GI algoritmi võib käsitleda kui hübriidi osakeste parve optimeerimise (PSO, *particle swarm optimization*) ja gravitatsioonilise optimeerimise meetodist, kus osakeste liikumistrajektoorid arvutatakse sarnaselt punktmasside liikumisvõrranditega gravitatsiooniväljas. Algoritmi tööd näidatakse kahe muutuja diofantilise võrrandi lahendamisel. Ühtlasi saab seejuures visualiseerida ja analüüsida otsimisruumi ning algoritmi käitumist kahemõõtmelisel tasandil.

Publication C

Spichakova, Margarita (2013). ***An approach to inference of finite state machines based on a gravitationally-inspired search algorithm.*** *Proceedings of the Estonian Academy of Sciences, 62(1), 39 – 46.*

Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 39–46
doi: 10.3176/proc.2013.1.05
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

An approach to the inference of finite state machines based on a gravitationally-inspired search algorithm

Margarita Spichakova

Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia; margo@cs.ioc.ee

Received 31 August 2011, revised 9 April 2012, accepted 21 September 2012, available online 20 February 2013

Abstract. As the inference of a finite state machine from samples of its behaviour is NP-hard, heuristic search algorithms need to be applied. In this article we propose a methodology based on applying a new gravitationally-inspired heuristic search algorithm for the inference of Moore machines. Binary representation of a Moore machine, an evaluation function, and the required parameters of the algorithm are presented. The experimental results show that this method has a lot of potential.

Key words: finite state machine, gravitational search algorithm, system identification.

1. INTRODUCTION

Identification is an inference process, which deduces an internal representation of a system (named *internal model*) from samples of its functioning (named *external model*) [1]. The inference of finite state machines (FSMs) is widely applied in different fields, such as logical design, verification, and software systems.

The goal of identification is to find the 'best' FSM, which respects the dynamics of the external model. In practice, the 'best' FSM is the one that best describes the model behaviour given by input–output sequences. We are interested in finding a minimum size deterministic FSM consistent with the set of the given samples. This is an NP-hard problem [2]. Heuristic algorithms are an alternative that can reduce the complexity of the identification methods.

The paper is organized as follows. Section 2 provides an overview of the problem of FSM inference. Section 3 describes gravitationally-inspired search algorithms. Section 4 introduces our approach, and Section 5 shows experimental results of the work.

2. INFERENCE OF FINITE STATE MACHINES

2.1. Problem statement

We give a brief overview of our approach to FSM identification. There are several types of FSMs, but in this article we will discuss only one well-known representation of them, namely the Moore machines.

A *Moore machine* is a six-tuple Mo $= \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$, where

- Q is a finite set of states, where q_0 denotes the initial state,
- Σ is the input alphabet,
- Δ is the output alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $\lambda : Q \to \Delta$ is the output function represented by the output table that shows what character from Δ will be printed by each state that is entered [3].

The general structure of our approach to the inference of FSMs is presented in Fig. 1.

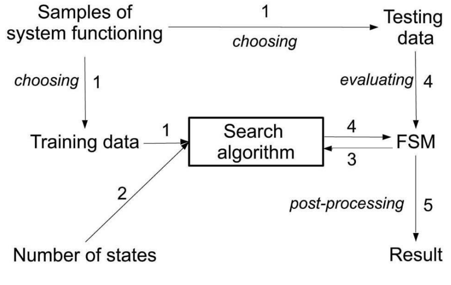


Fig. 1. Problem of system identification.

The outline of our approach is the following:

1. The system to be inferred is tested and samples of its functioning are generated. Some of the samples are chosen as *training data* and some as *testing data*.
2. The number of states in the FSM is received as an input.
3. The search algorithm is applied and a FSM M is outputted.
4. M is evaluated using the given training data and/or testing data. If M describes the given input–output data sufficiently well, it is considered as result. Otherwise the search process with other parameters or training data will be repeated.
5. If required, post-processing (e.g., minimization, reduction of unreachable states) is applied.

It is possible to specify several criteria for the required result. The first criterion is consistency of the FSM. Using this criterion, we can define two different types of solutions: the *generalized solution* (i.e., the solution that performs correctly for all positive input–output sequences) and the *consistent solution* (i.e., the solution that performs correctly for the input–output sequences used in the training set). Another criterion is the FSM size. We can search for the minimal FSM or a FSM with k or fewer states.

We formulate our goal as the inference of a *deterministic FSM with k or fewer states, consistent with input–output sequences at hand.*

2.2. Background

Heuristic techniques are widely applied to the inference of different types of FSMs. The most popular are the various types of Evolutionary Algorithms. In the early 1960s Fogel et al. [4] introduced Evolutionary Programming (EP). The simulated evolution was performed by modifying a population of FSM. Other authors also used EP for solving the problem of FSM identification. Chellapilla and Czarnecki [5] proposed the variation of EP to solve the problem of modular FSM synthesis. Benson [6] presented a model comprising an FSM with embedded genetic programs which co-evolve to perform the task of Automatic Target Detection.

Another approach to solve the problem of FSM identification is based on the Genetic Algorithm (GA). This method has been researched by several authors. Ngom et al. [7] used genetic simulation for Moore machine identification, Tongchim and Chongstitvatana [8] investigated parallel implementation of the GA to solve the problem of FSM synthesis. Lucas [9] paid more attention to finite state transducers and he and Reynolds [10] compared this method to 'Heuristic State Merging'. Niparnan and Chongstitvatana [11] improved GA by evolving only the state transition function. Chongstitvatana and Aporntewan [12] presented a method of FSM synthesis from multiple partial input/output sequences. Horihan and Lu [13] focused on improving the FSM evolution by using progressive fitness functions. Also Generated Simulated Annealing was used for the inference of FSM [14].

We apply a gravitationally-inspired search algorithm. The next section describes the general ideas of this new class of algorithms.

3. GRAVITATIONALLY-INSPIRED SEARCH ALGORITHM

3.1. Gravity as inspiration for heuristic search algorithms

Four main forces are acting in our universe: gravitational, electromagnetic, weak nuclear, and strong nuclear. These forces define the way our universe behaves and appears. The weakest force is gravitational; it defines how objects move depending on their mass. In physics three kinds of masses can be distinguished (active mass M_{a}, passive mass M_{p}, and inertial mass M_{i}), which have been shown experimentally to be equivalent (see [15]).

The gravitational force between two objects i and j is directly proportional to the product of their masses and inversely proportional to the square distance between them

$$F_{ij} = G\frac{M_{\mathrm{a}j} \cdot M_{\mathrm{p}i}}{R_{ij}^2}. \qquad (1)$$

Knowing the force acting on a body we can compute acceleration as

$$a_i = \frac{F_i}{M_{\mathrm{i}i}}. \qquad (2)$$

Our universe is growing, this yields an effect of decreasing gravity, so the gravitational 'constant' can be described as

$$G(t) = G(t_0) \cdot \left(\frac{t_0}{t}\right)^{\beta}, \beta < 1. \qquad (3)$$

We can formulate the following basic ideas inspired by gravity:

- Each object in the universe has mass and position.
- There are some interactions between objects, which can be described using the law of gravity.
- Bigger objects create larger gravitational fields and attract smaller ones.

During the last decade some researchers have tried to adapt the idea of gravity to find out optimal search algorithms. Such algorithms have some general ideas in common:

- The system is modelled by objects with mass.
- The position of those objects describes the solution, and the mass of the objects depends on the objective function.
- The objects interact with one another using gravitational force.
- The objects with greater mass present the points in the search space with better solutions.

Using these characteristics, it is possible to define the family of optimization algorithms based on gravitational force. For example, *Central Force Optimization* (CFO) is a deterministic gravity-based search algorithm proposed and developed by Formato [16]. It simulates the group of probes that fly into search space and explore it. Another algorithm, *Space Gravitational Optimization* (SGO), was developed by Hsiao et al. [17] in 2005. It simulates asteroids flying through curved search space. A gravitationally-inspired variation of local search, *Gravitational Emulation Local Search Algorithm* (GELS), was proposed by Webster and Bernhard [18] and further elaborated by Webster [19]. The newest one, *Gravitational Search Algorithm* (GSA), was described by Rashedi et al. [20] as a stochastic variation of CFO.

The next subsection will give a more detailed overview of the GSA, which is used as a basis of our approach.

3.2. Gravitational search algorithms

The GSA was described by Rashedi et al. [20] as a stochastic variation of the CFO and used for different applications. It was successfully applied to optimize various continuous problems, such as filter modelling [21], the set covering problem [22], allocation of static var compensator [15], and synthesis of thinned scanned concentric ring array antenna [23].

The algorithm is constructed so that there is a system of N objects, each of which is described by a real-valued position vector, and each position vector codes candidate solution

$$X_i = \left(x_i^1,\ldots,x_i^d,\ldots,x_i^n\right),\, d \in [1\ldots n], \tag{4}$$

where x_i^d represents the position of the ith object in dimension d.

Masses of objects are computed based on the quality measure as follows:

$$M_{\mathrm{a}i} = M_{\mathrm{p}i} = M_{\mathrm{i}i} = M_i,\, i \in [1,2,\ldots N], \tag{5}$$

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^N m_j(t)},\; m_i = \frac{fit_i(t) - worst(t)}{best(t) - worst(t)}, \tag{6}$$

where $worst(t)$ and $best(t)$ are defined for maximization problem as

$$best(t) = \underbrace{\max}_{j\in[1\ldots N]} fit_j(t),\; worst(t) = \underbrace{\min}_{j\in[1\ldots N]} fit_j(t),$$

and fit_i is the value of the objective function.

In other words, a heavier mass means that the quality of the object is better and it has greater attraction and inertia (i.e., moves slowly towards other objects).

At a specific time t we can recompute the force that is applied to the object i with mass M_i by some object j with mass M_j

$$F_{ij}^d(t) = G(t)\frac{M_{\mathrm{p}i}(t)\cdot M_{\mathrm{a}j}(t)}{R_{ij}+\varepsilon}(x_j^d - x_i^d), \tag{7}$$

where ε is a free parameter, required to avoid division by zero, and R_{ij} is the Euclidean distance between position vectors:

$$R_{ij} = \left\|X_i(t), X_j(t)\right\|. \tag{8}$$

According to Rashedi et al. [15], R_{ij} gives better experimental results than R_{ij}^2.

The gravitational constant G (Eq. (3)) is computed as

$$G(t) = G(G_0, t). \tag{9}$$

In physics, the general force acting on an object is computed as a vector sum of all acting forces. In the GSA, a stochastic characteristic is added to the algorithm, so the general force is computed as

$$F_i^d(t) = \sum_{j=1, i\neq j}^{N} rand_j \cdot F_{ij}^d(t),\; rand_j \in [0,1]. \tag{10}$$

The acceleration of object i can be computed knowing its inertial mass $M_{\mathrm{i}i}$ and force $F_i^d(t)$ as

$$a_i^d(t) = \frac{F_i^d(t)}{M_{\mathrm{i}i}(t)}. \tag{11}$$

Knowing current acceleration, we can recompute velocity and position as follows:

$$v_i^d(t+1) = rand_i v_i^d(t) + a_i^d(t),\; rand_i \in [0\ldots 1]; \tag{12}$$

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1). \tag{13}$$

The general procedure of the GSA is described in Algorithm 1. Firstly, the initial set of objects is generated randomly. Secondly, each object is evaluated. Based on evaluation results, the required parameters ($G(t)$, $worst(t)$, $best(t)$) are updated, and the forces and accelerations are computed. Thirdly, the agents' positions are changed according to acting forces and the updated positions are evaluated. The process continues until the best solution is found or the number of iterations is over.

Algorithm 1. General procedure of GSA

```
Generate initial positions
repeat
  Evaluate quality of each object
  Update $G(t)$, $worst(t)$, $best(t)$
  Calculate masses and accelerations
  Calculate velocities and positions
until meeting ending criterion
Return best solution
```

In the GSA, the position vector is real-valued. However, for some applications discrete or binary vectors are required. A discrete modification of the algorithm was proposed by Zibanezhad et al. [24] in a context of Web-Service composition. The binary GSA (BGSA) was introduced by Rashedi et al. [25] in 2010. In the next section we will focus on the BGSA.

3.3. Binary gravitational search algorithm

The key difference between the GSA and the BGSA is the binary search space, meaning that each dimension has only two possible values: '0' or '1'. The main laws of the BGSA may be defined as in real-valued case (see Eqs (7), (11), and (12)). But the positions' updating law (see Eq. (13)) must be modified so that each dimension changes between two values according to the velocity. A higher velocity gives a greater probability of changing the value.

To modify Eq. (13), in the BGSA a special probability function $S(v_i^d)$ was introduced, which transfers the value of v_i^d to $[0 \dots 1]$:

$$S(v_i^d) = \left|\tanh(v_i^d)\right|. \tag{14}$$

The law for updating the position can be defined as follows:

$$x_i^d(t+1) = \begin{cases} F(x_i^d(t)) & \text{if } rand < S(v_i^d(t+1)), \\ x_i^d(t) & \text{if } rand \geq S(v_i^d(t+1)), \end{cases} \tag{15}$$

where $F(x_i^d(t)) = complement(x_i^d(t))$. Some other modifications were made:

- Velocity v_i^d is bounded: $\left|v_i^d\right| < v_{\max}$.
- Distance R is computed as the Hamming distance.
- Gravitational constant G is considered as a linear decreasing function

$$G(t) = G_0(1 - t/T). \tag{16}$$

4. GRAVITATIONALLY-INSPIRED SEARCH ALGORITHM FOR THE INFERENCE OF FSMs

To apply the BGSA to the inference of FSMs we need to define an objective function and a process of encoding FSMs to a binary position vector. Also modifications of the original BGSA have to be made.

4.1. Representation of an FSM

We discuss only Moore machines with exactly n states. Consider a target machine Mo with n states, input alphabet $\Sigma = \{i_0, \dots, i_{l-1}\}$, output alphabet $\Delta = \{o_0, \dots, o_{m-1}\}$, and set of states $Q = \{q_0, \dots, q_{n-1}\}$.

To store the information about state q_j, we need to store the output value o^j of the state and corresponding transitions from the given state q_j to get some target state q^{i_k}, which are activated by reading symbol i_k. Each section represents one state (Fig. 2), where the first part is an output value of the state and the other part stores the corresponding transitions from that state. Initially, information is presented in a decimal way (decimal representation). To get binary representation we transform each integer number to the corresponding binary number.

The number of bits required for storing the whole binary position vector can be computed as follows:

$$Length = n \cdot (\lceil \log_2 m \rceil + l \lceil \log_2 n \rceil). \tag{17}$$

Each Mo has a unique binary representation, but not each binary string has a corresponding Mo.

Let us take a look at a Moore machine with the transition diagram presented in Fig. 3.

We have four states $Q = \{0, 1, 2, 3\}$, the input alphabet contains two symbols $\Sigma = \{a, b\}$, and the output alphabet two symbols $\Delta = \{0, 1\}$.

Thus we need 20 bits to store this FSM (Eq. (17)): $4 \cdot (\lceil \log_2 2 \rceil + \lceil \log_2 4 \rceil \cdot 2) = 20$ bits. The general structure of the position vector required to encode this FSM is presented in Fig. 4.

State q_j			
o^j	q^{i_0}	$q^{\cdots}$	$q^{i_{k-1}}$

Fig. 2. A section of the binary position vector for storing the Moore machine with a fixed number of states.

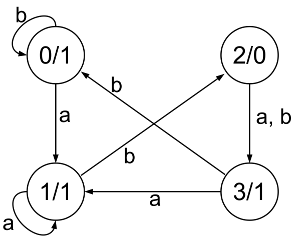


Fig. 3. A Moore machine represented as a transition diagram.

	a	b		a	b		a	b		a	b	
1	1	0	1	1	2	0	3	3	1	1	0	Dec. representation
1	01	00	1	01	10	0	11	11	1	01	00	Bin. representation

Fig. 4. Example. Binary position vector for storing the Moore machine.

4.2. An objective function

We propose an objective function defined on all input–output sequences (pairs {input, output}). The idea is to estimate the proximity between the current and the desired FSMs by finding the distance between strings.

4.2.1. Distance between strings

Consider a function $\Delta(a,b)$, where a,b are symbols in some alphabet, and define

$$\Delta(a,b) = \begin{cases} 0 & : a = b, \\ 1 & : a \neq b. \end{cases} \tag{18}$$

That is, if character a is not equal to character b, the function $\Delta(a,b)$ will return 1, otherwise the function will return 0.

We propose two distance functions between strings x and y. The first function is the *Hamming distance* d_{Ham}. To compute it, we need to count the number of different bits in the same positions

$$d_{\mathrm{Ham}}(x,y) = \Sigma_{i=1}^{\min(|x|,|y|)} \Delta(x_i,y_i). \tag{19}$$

The second function evaluates the *length of maximal equal prefix* d_{LP} (i.e., the computation will be stopped at the first difference between strings)

$$d_{\mathrm{LP}}(x,y) = \Sigma_{i=1}^{x=y} \Delta(x_i,y_i). \tag{20}$$

4.2.2. Evaluation of the objective function

We specify several objective functions for evaluating FSMs based on d_{Ham} and d_{LP}. Assume we have our training data represented as a collection of input–output sequences (the size of the collection is n). We also have output strings produced by an FSM (see Table 1).

Our task is to measure how ‘far’ the strings generated by the FSM are from the expected strings. The objective function based on the Hamming distance (d_{Ham}) defines the *objective function* as follows:

$$OF = \Sigma_{i=1}^{n} \left(l_i - d_{\mathrm{Ham}} \left(\mathrm{Out}_i^{\mathrm{expected}}, \mathrm{Out}_i^{\mathrm{produced}} \right) \right), \tag{21}$$

where n is the number of the given data and l_i is the length of $\mathrm{Out}_i^{\mathrm{expected}}$.

Table 1. Measuring the value of the objective function

Input	Expected output	Produced output	Distance
In_0	$\mathrm{Out}_0^{\mathrm{expected}}$	$\mathrm{Out}_0^{\mathrm{produced}}$	$\mathrm{dist.}_0$
In_1	$\mathrm{Out}_1^{\mathrm{expected}}$	$\mathrm{Out}_1^{\mathrm{produced}}$	$\mathrm{dist.}_1$
...	...	...	...
In_n	$\mathrm{Out}_n^{\mathrm{expected}}$	$\mathrm{Out}_n^{\mathrm{produced}}$	$\mathrm{dist.}_n$

In the second case we use d_{LP} for measuring the distance. Thus, the *objective function* can be defined as the sum of the lengths of all sequences

$$OF = \Sigma_{i=1}^{n} \left(d_{\mathrm{LP}} \left(\mathrm{Out}_i^{\mathrm{expected}}, \mathrm{Out}_i^{\mathrm{produced}} \right) \right). \tag{22}$$

4.3. Algorithm description

In this section we will focus on the properties of our algorithm. Search space is described by a set of binary position vectors, where each position vector corresponds to an FSM as described in Section 4.1.

First, we set
- the free parameter ε,
- the maximal speed $v_{\max}$,
- the number of iterations,
- the number of objects,
- the number of states n in the FSM,
- the initial value of gravitational constant G_0,
- the mass value minimum $M_{\min}$

according to the problem under consideration.

The initial positions are generated randomly from the feasible region, so that each position corresponds to an FSM. To do so, the FSM is generated in decimal form, a number of symbols in input and output alphabet are restored from the input data. After generating the FSM in decimal form it is encoded into binary representation (see Section 4.1).

The objective function of a candidate solution is computed as described in Subsection 4.2.2. Despite the fact that in physics the active, passive, and inertial masses are considered to be equivalent (see 3.1), we modified mass computation laws to improve the search algorithm. The active M_{a}, passive M_{p}, and inertial M_{i} masses are computed as follows

$$M_{\mathrm{p}} = M_{\mathrm{i}} = \frac{OF}{OF_{\max}}, \tag{23}$$

$$M_{\mathrm{a}} = \begin{cases} M_{\mathrm{i}} & \text{if } M_{\mathrm{a}} > M_{\min}, \\ 0 & \text{if } M_{\mathrm{a}} \le M_{\min}. \end{cases} \tag{24}$$

If M_{a} is smaller than the minimum value $M_{\min}$ of the defined mass, then $M_{\mathrm{a}} = 0$ (i.e., an object with a smaller mass does not create a gravitational field).

Forces acting on the object are computed via Eq. (7). The distance in one dimension can be computed as follows:

$$(x_j^d - x_i^d) = \begin{cases} 1 & \text{if } x_j^d \neq x_i^d, \\ -1 & \text{if } x_j^d = x_i^d. \end{cases} \tag{25}$$

The acceleration vector is computed via Eq. (11). The velocity vector is computed by Eq. (12). If the velocity is higher than $v_{\max}$, then its value will be set to $v_{\max}$.

The new position is computed using the old position and the velocity vector (see Eq. (15)). The probability function (i.e., the threshold function) $S(v_i^d)$ is taken as

$$S(v_i^d) = \left|\sin(v_i^d)\right|, \tag{26}$$

in this case $v_{\max} = \pi/2$.

5. IMPLEMENTATION AND EXPERIMENTS

Our approach was implemented in Java (JDK 1.5) and tested on random machines and some 'toy' examples. Results are compared to the canonical Genetic Algorithm (more details about GA can be found in [10,26]).

5.1. Experiments I

Experiments were constructed so that general parameters, such as the number of iterations and the number of objects, the encoding of the Moore machine, and its initialization algorithm are the same (see Subsection 4.1). Evaluation of the machine is described in Subsection 4.2; the objective function is constructed on the Hamming similarity. The specific parameters of the algorithm are described for a concrete experiment in the corresponding table.

During the experiments, each algorithm was run 20 times with a different initial set of objects. Results are presented in Table 2 and Table 3, where the row 'Init. %' shows the mass value of the best solution at the initial step (randomly generated), the row 'Sol. %' shows the object value of the best found solution, and the row 'Iter.' shows how many iterations were required to find this solution ('–' means that the best possible solution was not found).

5.1.1. Pattern recognizer

The goal of this experiment was to reconstruct a pattern 'aab' recognizer (see Table 2) from the given input–output pairs. As input data we use six pairs with each input string having a length of 12. The number of states n is four. The number of iterations is taken 100, and the number of objects equals 200.

This experiment showed that the BGSA was more frequently able to find 100% solutions than the GA (10/20 compared to 7/20 for GA) and fewer iterations were required to find them.

5.1.2. Parity checker

The goal of this experiment was to reconstruct a parity checker (see Table 3) from the given input–output pairs. As input data we use seven pairs with length 8 of each input string. The number of states n equals two. The number of iterations is taken 20, and the number of objects equals five.

This experiment showed that the BGSA was more frequently able to find 100% solutions than the GA (14/20 compared to 10/20 for GA) and fewer iterations were required to find them. In three out of twenty cases the GA was not able to improve the maximal solution that was randomly generated in the initial population; for the GSA this happened only in one case out of twenty.

Table 2. Experiment I.1 'Pattern recognizer'

	BGSA, $\varepsilon = 0.01$, $G_0 = 100$, $M_{\min} = 0.1$																
Init. %	92	94	92	95	87	92	89	87	91	87	92	89	89	96	92	86	98
Sol. %	100	99	99	100	99	100	100	100	100	100	100	100	98	98	100	98	98
Iter.	66	–	–	2	–	32	70	14	2	38	2	75	–	–	1	–	–
	GA, $P_{\text{mutation}} = 0.04$, roulette wheel selection, one point crossover																
Init. %	98	87	87	95	87	87	87	87	92	96	87	87	92	93	87	87	92
Sol. %	100	100	100	95	100	92	92	91	100	96	100	96	94	99	91	92	96
Iter.	75	13	65	–	52	–	–	–	14	–	82	–	–	–	–	–	–

Table 3. Experiment I.2 'Parity checker'

	BGSA, $\varepsilon = 0.01$, $G_0 = 10$, $M_{\min} = 0.1$																
Init. %	49	55	65	65	55	55	55	55	55	55	65	55	61	55	57	55	55
Sol. %	100	65	100	65	65	100	100	65	100	100	100	65	100	100	65	100	100
Iter.	1	–	9	–	–	11	15	–	14	14	6	–	9	7	–	15	1
	GA, $P_{\text{mutation}} = 0.04$, roulette wheel selection, one point crossover																
Init. %	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55
Sol. %	100	65	100	55	100	65	100	100	100	65	100	65	65	100	100	55	55
Iter.	12	–	11	–	8	–	12	7	5	–	15	–	–	13	5	–	–

5.2. Experiments II

The goal of those experiments was to compare the BGSA and GA for the same random initial set of objects. Tasks were taken as in the previous experiments (i.e., 'pattern recognizer' and 'parity checker'). Those experiments were constructed in such a way that the initial population was taken the same for both algorithms, the parameters such as the number of iterations and the number of objects were also equal for both algorithms, and are described in Subsection 5.1. Each algorithm (BGSA and GA) was executed 10 times with the same initial set of objects as in Experiments I (5.1). The average best-so-far solutions are presented in Fig. 5. According to the results of this experiment, in the case of 'pattern recognizer' the BGSA solves the task better than the GA (Fig. 5a). For the second task, 'parity checker' (Fig. 5b), the BGSA behaves almost like the GA.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a method for the inference of Moore machines based on a gravitationally-inspired search algorithm. Binary representation of FSMs and different types of objective functions were introduced. Parameters and variations of the proposed algorithm were discussed. The proposed approach was implemented and successfully tested using random data and different examples. During the first experiments, our approach gave promising results.

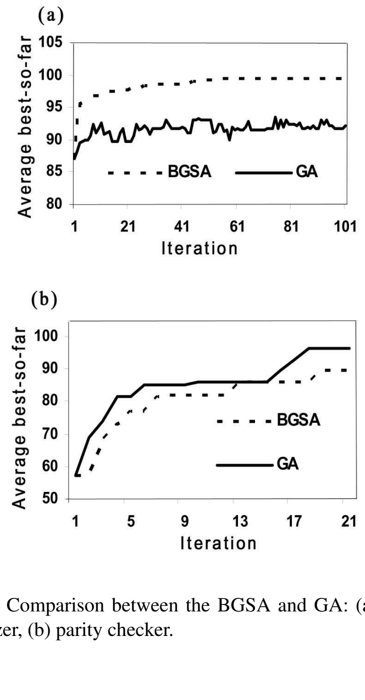


Fig. 5. Comparison between the BGSA and GA: (a) pattern recognizer, (b) parity checker.

To improve the quality of the proposed approach, parameters of the algorithm and their effect on the presented methods will be explored. The effect of using different aspects of laws will be investigated. During further developments the proposed method will be adjusted to take into account other types of FSM, for example the Mealy machines.

ACKNOWLEDGEMENTS

This research was supported by the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

REFERENCES

1. Angluin, D. and Smith, C. H. Inductive inference: theory and methods. *ACM Comput. Surv.*, 1983, **15**, 237–269.
2. Gold, E. M. Complexity of automaton identification from given data. *Inform. Control*, 1978, **37**(3), 302–320.
3. Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. International Edition (2nd edn). Addison-Wesley, 2003.
4. Fogel, L. J., Owens, A. J., and Walsh, M. J. *Artificial Intelligence Through Simulated Evolution*. Wiley, Chichester, UK, 1966.
5. Chellapilla, K. and Czarnecki, D. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the 1999 Congress on Evolutionary Computation*. Vol. 2. IEEE Press, 1999, 1349–1356.
6. Benson, K. A. Evolving finite state machines with embedded genetic programming for automatic target detection within SAR imagery. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*. IEEE Press, 2000, 1543–1549.
7. Ngom, L., Baron, C., and Geffroy, J. Genetic simulation for finite state machine identification. In *SS '99: Proceedings of the Thirty-Second Annual Simulation Symposium*. IEEE Computer Society, Washington, DC, USA, 1999, 118.
8. Tongchim, S. and Chongstitvatana, P. Parallel genetic algorithm for finite state machine synthesis from input/output sequences. In *Evolutionary Computation and Parallel Processing* (Cantu-Paz, E. and Punch, B., eds). Las Vegas, Nevada, USA, 2000, 20–25.
9. Lucas, S. M. Evolving finite state transducers: some initial explorations. In *EuroGP*. 2003, 130–141.
10. Lucas, S. M. and Reynolds, T. J. Learning finite state transducers: evolution versus heuristic state merging. *IEEE T. Evolut. Comput.*, 2007, **7**, 308–325.
11. Niparnan, N. and Chongstitvatana, P. An improved genetic algorithm for the inference of finite state machine. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002, 189.

12. Chongstitvatana, P. and Aporntewan, C. Improving correctness of finite-state machine synthesis from multiple partial input/output sequences. In *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware*. 1999, 262–266.
13. Horihan, J. W. and Lu, Y.-H. Improving fsm evolution with progressive fitness functions. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes Symposium on VLSI*. ACM Press, New York, NY, USA, 2004, 123–126.
14. Cerruti, U., Giacobini, M., and Liardet, P. Prediction of binary sequences by evolving finite state machines. In *Selected Papers from the 5th European Conference on Artificial Evolution*. Springer-Verlag, London, UK, 2002, 42–53.
15. Rashedi, E., Nezamabadi-pour, H., Saryazdi, S., and Farsangi, M. M. Allocation of static var compensator using gravitational search algorithm. In *First Joint Congress on Fuzzy and Intelligent Systems, Ferdowsi University of Mashhad, Iran, 29–31 August, 2007*, 29–31.
16. Formato, R. A. Central force optimization: a new metaheuristic with applications in applied electromagnetics. *PIER*, 2007, **77**, 425–491.
17. Hsiao, Y.-T., Chuang, C.-L., Jiang, J.-A., and Chien, C.-C. A novel optimization algorithm: space gravitational optimization. In *IEEE International Conference on Systems, Man and Cybernetics, 2005*, Vol. 3. 2005, 2323–2328.
18. Webster, B. and Bernhard, P. J. A local search optimization algorithm based on natural principles of gravitation. Technical Report CS-2003-10, Florida Institute of Technology, 2003.
19. Webster, B. *Solving Combinatorial Optimization Problems Using a New Algorithm Based on Gravitational Attraction*. PhD thesis, Florida Institute of Technology, Melbourne, FL, USA, 2004.
20. Rashedi, E., Nezamabadi-pour, H., and Saryazdi, S. GSA: a gravitational search algorithm. *Inform. Sciences*, 2009, **179**(13), 2232–2248.
21. Rashedi, E., Nezamabadi-pour, H., and Saryazdi, S. Filter modeling using gravitational search algorithm. *Eng. Appl. Artif. Intell.*, 2011, **24**, 117–122.
22. Balachandar, S. R. and Kannan, K. A meta-heuristic algorithm for set covering problem based on gravity. *International Journal of Computational and Mathematical Sciences*, 2010, **4**(5), 223–228.
23. Chatterjee, A., Mahanti, G. K., and Pathak, N. Comparative performance of gravitational search algorithm and modified particle swarm optimization algorithm for synthesis of thinned scanned concentric ring array antenna. *PIER B*, 2010, **25**, 331–348.
24. Zibanezhad, B., Zamanifar, K., Nematbakhsh, N., and Mardukhi, F. An approach for web services composition based on QoS and gravitational search algorithm. In *Proceedings of the 6th International Conference on Innovations in Information Technology, IIT'09*. IEEE Press, Piscataway, NJ, USA, 2009, 121–125.
25. Rashedi, E., Nezamabadi-pour, H., and Saryazdi, S. BGSA: binary gravitational search algorithm. *Nat. Comp.*, 2010, **9**, 727–745.
26. Fabera, V., Janes, V., and Janesova, M. Automata construct with genetic algorithm. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*. IEEE Computer Society, Washington, DC, USA, 2006, 460–463.

Meetod lõplike automaatide genereerimiseks gravitatsiooniseadusest inspireeritud otsimisalgoritmi abil

Margarita Spichakova

Kuna lõplike automaatide genereerimine sisend-väljundpaaride näidiste alusel on NP-keerukas ülesanne, tuleb selle lahendi leidmiseks kasutada heuristilisi algoritme. Artiklis on pakutud metoodika Moore'i masinate genereerimiseks, kasutades uut, gravitatsiooniseadusest inspireeritud otsimisalgoritmi. On esitatud algoritmi rakendamiseks vajalik Moore'i masina binaaresitus, sihifunktsioon ja algoritmi juhtimiseks kasutatavad parameetrid. Eksperimendid näitavad, et lähenemisel on arvestatav potentsiaal.

CURRICULUM VITAE

Personal data

Name:	Margarita Spichakova
Date of birth:	10.08.1983
Place of birth:	Tallinn, Estonia

Contact data

Phone:	+372 504 5503
E-mail:	margarita.spitsakova@ttu.ee

Education

2007 – ... :	Tallinn University of Technology	PhD studies
2005 – 2007:	Tallinn University of Technology	M.Sc. in Informatics
2001 – 2005:	Tallinn University of Technology	B.Sc. in Informatics

Language competence

Russian:	Native speaker
Estonian:	Fluent
English:	Fluent

Professional employment

2017 – ... :	Tallinn University of Technology, Department of Software Science	Early-stage researcher
2015 – 2016 :	Tallinn University of Technology, Institute of Cybernetics	Engineer (0.25)
2010 – 2015 :	Tallinn University of Technology, Institute of Cybernetics	Engineer (1.0)
2007 – 2010 :	Tallinn University of Technology, Institute of Cybernetics	Extraordinary Researcher (1.0)
2005 – 2007 :	Tallinn University of Technology, Institute of Cybernetics	Engineer (1.0)

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Margarita Spitšakova
Sünniaeg:	10.08.1983
Sünnikoht:	Tallinn, Eesti

Kontaktandmed

Telefon:	+372 504 5503
E-post:	margarita.spitsakova@ttu.ee

Hariduskäik

2007 – ... :	Tallinna Tehnikaülikool	Doktorantuur
2005 – 2007:	Tallinna Tehnikaülikool	M.Sc., Informaatika
2001 – 2005:	Tallinna Tehnikaülikool	B.Sc., Informaatika

Keelteoskus

Vene keel:	Emakeel
Eesti keel:	Kõrgtase
Inglise keel:	Kõrgtase

Teenistuskäik

2017 – ... :	Tallinna Tehnikaülikool, Tarkavarateaduse Instituut	Doktorant-nooremteadur
2015 – 2016 :	Tallinna Tehnikaülikool, Küberneetika Instituut	Engineer (0.25)
2010 – 2015 :	Tallinna Tehnikaülikool, Küberneetika Instituut	Engineer (1.0)
2007 – 2010 :	Tallinna Tehnikaülikool, Küberneetika Instituut	Erakorraline teadur (1.0)
2005 – 2007 :	Tallinna Tehnikaülikool, Küberneetika Instituut	Engineer (1.0)

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-pin-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and Physical Modelling of Dynamic Electrical Impedance. 2016.

115. **Aivo Anier**. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems. 2016.

116. **Denis Firsov**. Certification of Context-Free Grammar Algorithms. 2016.

117. **Sergei Astapov**. Distributed Signal Processing for Situation Assessment in Cyber-Physical Systems. 2016.

118. **Erkki Moorits**. Embedded Software Solutions for Development of Marine Navigation Light Systems. 2016.

119. **Andres Ojamaa**. Software Technology for Cyber Security Simulations. 2016.

120. **Gert Toming**. Fluid Body Interaction of Biomimetic Underwater Robots. 2016.

121. **Kadri Umbleja**. Competence Based Learning – Framework, Implementation, Analysis and Management of Learning Process. 2017.

122. **Andres Hunt**. Application-Oriented Performance Characterization of the Ionic Polymer Transducers (IPTs). 2017.

123. **Niccolò Veltri**. A Type-Theoretical Study of Nontermination. 2017.

124. **Tauseef Ahmed**. Radio Spectrum and Power Optimization Cognitive Techniques for Wireless Body Area Networks. 2017.