Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Informatics

ADAPTIVE CONTROL ALGORITHMS OF INTELLIGENT AGENTS

by

MARTIN ŠLAPÁK

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree Doctor.

Dissertation degree study programme: Informatics

Prague, March 2019

SUPERVISOR:

Mgr. Roman Neruda, CSc.
Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2
18207 Prague 8
Czech Republic

This thesis is dedicated to all little dragons
and
to whoever might find it interesting…

ABSTRACT

The multi-agent systems and agent-based approach to complex and non-trivial practical problems have become more popular in recent years. The traditional methods of agent's control sometimes fail or are inappropriate. The adaptive methods seem to be a possible solution for these cases. This thesis deals with an exploration of possibilities of design and development of adaptive methods of intelligent software agent's control. These agents solve problems from data mining domain. The thesis brings an overview of given problem areas, describes experiments, and their results. In conclusions, the topics of author's future research are proposed.

ABSTRAKT

V posledních letech se dostávají do popředí zájmu multiagentní systémy a agentní přístup k řešení komplexních a netriviálních praktických problémů. Ne vždy je možné či vhodné takovéto softwarové systémy řídit tradičními a pevně danými způsoby řízení. Své místo tak dostávají adaptivní metody. Tato práce si klade za cíl prozkoumat možnosti návrhu a vývoje adaptivních metod pro řízení inteligentních softwarových agentů řešících úlohy z prostředí data miningu. Práce přináší přehled problematiky, popisuje uskutečněné experimenty a jejich výsledky. V závěru jsou navržena témata dalšího výzkumu autora.

PUBLICATIONS

This is a list of author's publication related to this thesis. Some ideas and figures have appeared previously in the following publications:

[1] R. Neruda and M. Šlapák. "Evolving decision strategies for computational intelligence agents." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7390 LNAI (2012), pp. 213–220. DOI: 10.1007/978-3-642-31576-3_28.

[2] M. Šlapák. "Genetics in decision behavior of computational agents." In: *Mendel* (2011), pp. 44–47.

[3] M. Šlapák and R. Neruda. "Multiobjective Genetic Programming of Agent Decision Strategies." In: *Advances in Intelligent Systems and Computing* 303 (2014), pp. 173–182. DOI: 10.1007/978-3-319-08156-4_18.

[4] M. Šlapák and R. Neruda. "Tree based decision strategies and auctions in computational multi-Agent systems." In: *Investigacion Operacional* 38.4 (2017), pp. 335–342. ISSN: 0257-4306.

[5] M. Šlapák and R. Neruda. "Matching subtrees in genetic programming crossover operator." In: *ICNC-FSKD 2017 - 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery* (2018). Ed. by et al. Zhao L., pp. 208–213. DOI: 10.1109/FSKD.2017.8393091.

The laws of genetics apply even if you refuse to learn them.

— Alison Plowden

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my dissertation thesis supervisor, *Roman Neruda*. He has been a source of my passion for evolutionary algorithms. He guided my first steps on the researcher way. Without his willingness to work regardless of the days of the week(end) most of our publications and this thesis would be hardly possible.

Special thanks go to the staff of the Department of Theoretical Informatics, apart from others, especially to *Pavel Kordík* for a lot of helpful advice and insight to the faculty life.

Many thanks go to my amazing wife *Míša* for patience and never ending support during my entire PhD studies, proof reading, language corrections. She made this time happier and inspired me with her approach to the research.

I would like to express thanks to my colleagues from room A1246, namely to *Jan Trávníček*, *Radomír Polách* and others for a friendly environment and sharing funny moments of PhD studies. Acknowledgement also goes to members of Computer Intelligence Group at FIT CTU and of course to the colleagues "Roman's children" from MFF CUNI for their valuable comments on my research.

I would also like to thank *André Miede* the author of this bright LaTeX thesis template which is provided for free. It is precisely prepared and has a perfect formed output.

This research has been partially supported by Institutional support (RVO18000), by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM 6840770014 and internal grants from the CTU in Prague – SGS No. SGS11/083/OHK3/1T/-18, No. SGS12/096/OHK3/1T/18, No. SGS14/102/OHK3/1T/18, and No. SGS17/210/OHK3/3T/18.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AI	Artificial Intelligence
ANN	Artificial Neural Networks
API	Application Programming Interface
AVA	Average Arity
BDI	Belief Desire Intention
BDI	Belief, Desire, Intention
COS	Common Operator Set
CTL	Computation Tree Logic
CVS	Common Variable Set
EA	Evolutionary Algorithms
GP	Genetic Programming
GUI	Graphical User Interface
JADE	Java Agent Development Framework
KDD	Knowledge Discovery in Data
KQML	Knowledge Query and Manipulation Language
LTL	Linear Temporal Logic
MAS	Multiagent Systems
MCO	Multi-criteria Optimization
SCO	Single-criteria Optimization
MSE	Mean Squared Error
OOP	Object Oriented Programming
OS	Operating System

RBF	Radial Basis Function
REV	Recurrent Evaluation
STR	String Similarity
VEV	Vector Evaluation

Part I

INTRODUCTION

1

INTRODUCTION

1.1 MOTIVATION

Today we are surrounded by many automated distributed systems. These systems often consist of smaller parts which can cooperate or compete. Such systems meet the definition of multi-agents systems [93].

multi-agent systems

The Multiagent Systems (MAS) are invented and applied to a broad spectrum of problems of research, and also in industrial tasks. The main reasons for the application of MAS are growing complexity of problems, efficient use of distributed resources, natural decomposition [66] and the insufficiency of traditional approaches.

In some cases, we are not able to specify a correct control routine for each agent by hand or some known deterministic algorithm. For such situations, we need to use some adaptive algorithms and techniques commonly used in artificial intelligence – e. g. reinforcement learning, evolutionary algorithms, neural networks or genetic programming.

In this thesis, we focus on the potential of Genetic Programming (GP) to evolve efficient adaptive control algorithms for task distribution in MAS.

1.2 THE GOAL OF THE PRESENTED THESIS

The goal is to explore the capability of GP to evolve adaptive control mechanisms for the problem of *bottom-up* oriented decision making in task allocation in a data-mining MAS.

The goal can be divided into the following sub-goals:

1. *Explore the possibilities to control agents in MAS by decision trees in simple experimental setup.* We verify evolved trees as a control mechanism in the experimental setup with small artificial tasks being solved concerning one simple criterion. The number of task types and models will be limited to a minimal amount. The combination of the average processing time of each task and the total time of the experiment run will be used as the criterion.

2. *Investigate the feasibility to use GP evolved trees on complex tasks.* We replace the simple artificial tasks from the first experiment

with real ones which are more complex. These tasks have larger datasets and will be solved by real data-mining models (classification and regression). If necessary, the set of worker agent's attributes and also the task's attributes will be revised. We stay at Single-criteria Optimization (SCO). As a criterion, the mean error of the processed task will be used.

3. *Extend the SCO to use multiple criteria.* For that criteria the task set will be solved by agents in MAS. We try to combine both criteria (mean error and time) and use SCO. The next natural step is the utilization of Multi-criteria Optimization (MCO). For that, we use the NSGA-II algorithm [19].

4. *Explore the possibilities and propose an improvement of crossover operator in GP.* The crossover operator in GP is known as possible source of bloating problem [56, 78]. We focus on the searching of the best method for matching subtrees for the crossover operator. The verification and performance measures will be done on the problem of symbolic regression on several benchmark datasets.

5. *Compare tree-based control mechanisms with another task distribution control system.* We selected the task allocation driven by auction systems for comparison. The best-known subtree matching method and the configuration of evolution parameters from previous experiments will be used in the evolution. The experiment will be performed on real data mining tasks. The optimization criterion is a combination of mean error and average processing time for each task.

1.3 PROBLEM DEFINITION

This thesis focuses on *bottom-up* oriented task distribution and solving in data-mining MAS. Consider a data-mining system where a set of tasks is being processed in a distributed manner by individual units – agents encapsulating a specific data-mining model. The *bottom-up* approach gives the responsibility for allocation of a particular task to each of the processing units, not to the main (central) agent which is responsible for the solution of the whole task set on the other side. The task set has to be solved concerning a set of criteria. The minimal error or the shortest time aggregated over the whole task set represent good examples of such criteria.

The trees as an adaptive control mechanism evolved by GP.

The GP produces trees which are used as a control mechanism in processing units. All processing agents use the same decision tree. The tree combines properties of the offered task and also of the processing unit and controls the decision-making process whether the unit should accept or reject the offered task. To measure the performance of the evolved tree, it is necessary to run the experiment and process the whole task-set. Based on the results the fitness function of the individual tree is computed. The consequence is that the calculation of fitness is a time expensive process.

Tasks distribution and solving in MAS

There are two possible approaches to task set distribution. One of them is the central control where one entity directs the splitting of a task set and rules the whole process. This approach can be named as *top-down*. The central entity should know and take into account all additional knowledge about the task set which is being solved. In the other case, which can be called a *bottom-up* approach, the central entity divide the whole task set at random and the decision which part (particular task) is suitable for which method (encapsulated in the processing agent) lies on the solving units.

top-down and bottom-up approaches

We set the target for agents in MAS in the following way: We have a set of various data mining tasks (classification, regression) defined by its goal and input data. The whole task set has to be solved by our agents encapsulating different data mining models in MAS concerning one or more criteria. An example of a simple data mining task definition is shown in the listing 1. The set of instances of such data mining task definitions forms the *task set*. The number of instances differs in particular experiments from 100 to 1000 if not stated otherwise.

Listing 1: An example of the classification dataset as a part of a task set.

```
@RELATION iris

@ATTRIBUTE sepallength  REAL
@ATTRIBUTE sepalwidth   REAL
@ATTRIBUTE petallength  REAL
@ATTRIBUTE petalwidth   REAL
@ATTRIBUTE class        {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
...
7.0,3.2,4.7,1.4,Iris-versicolor
```

Adaptive agent control

Each agent has some mechanism which guides its steps. For a software agent, it is some method, or it may also be a whole complex system consisting of many parts. It may be *static* for the whole agent's life, or it may be adaptive and variable in different states of the agent's life.

Adaptive control can be achieved by many approaches and depends on the designer and selected architecture and also on the current implementation. From the global point of view, we can divide adaptive control into two classes – the first one contains such control systems which are adaptive continuously during the whole agent's live (system run). The second class associates control methods whose adaptivity means that there exist some attributes which are changing. These attributes are incorporated into the agent's decision-making process.

the roles of agents

In our data mining MAS we have two main types of agents. The first type is called a manager. This agent is responsible for distribution of all tasks from the given task set to the processing agents, for the collection of the results of solved tasks and for the redistribution of the tasks which are rejected by the processing agents. The second type is represented by a processing unit which is called a worker. A worker is responsible for deciding whether to accept or reject the offered task. If it decides to accept, it has to process the task and send the results back to the manager. In cases when the offered task looks perspective for given worker, it is possible to interrupt processing of the particular task and return interrupted task as unsolved. The perspective task will be solved instead of the returned one.

We defined the control process of the manager statically. It has hard-coded behavior which rules its actions during the experiment run. As stated above the tasks are distributed by the manager at random. The algorithm of tasks distribution is shown in listing 2.

Listing 2: A pseudocode of manager's task distribution. Both functions (*distribute, collect*) are executed in parallel

```
function distribute () {
  while(t ← get_fresh_task(task_set)) {
    a ← worker_agents[rnd()]
    send_task(t, a)
    t.fresh = False
  }
}

function collect () {
  while(t ← get_task_from(incomming_buffer)) {
    if (t.solved == True) {
      process_results(t)
    } else {
      t.fresh = True
      update(task_set, t)
    }
  }
}
```

The control process of a worker agent can be divided into two main parts. The first one is to determine whether the offered task should be accepted for processing or not. If not, it is returned to the manager with attribute *solved* set to *False*. Such a task is subsequently redistributed by the manager. If the given task is accepted, it is processed with the data mining model in the worker agent.

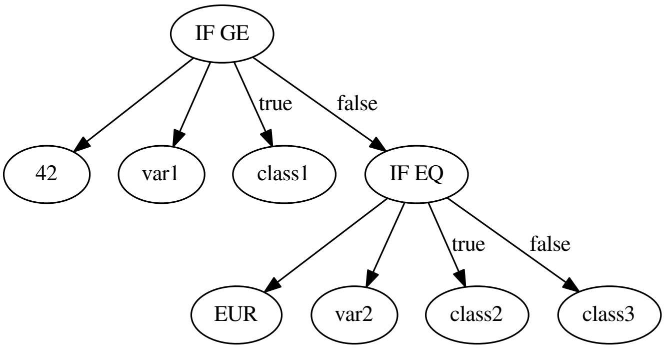


Figure 1: An example of small decision tree.

In the decision-making process of task accepting the decision trees are utilized. The concept of decision trees will be described in detail in chapter 6.1. Briefly, the decision tree connects a set of defined attributes of the environment (worker agent and offered task). An example of such a tree is depicted on the Fig. 1. When the tree is evaluated, the value representing some decision is returned. As the current value of attributes is changed during the processing of the whole task set the tree has to be capable of responding to these changes. In other words, the tree is adaptive.

Based on empirical results obtained during the experiments performed in this work we changed some methods and configurations. We started with decision trees which represent a polynomial structure. This structure connects weighted attributes, and the tree is evaluated from the leave nodes to the root. The value in the root node is then compared to a defined threshold, and this relation determines the final decision. A closer description of this approach can be found in sections 7.1 and 7.2.

Later we introduced the *if* inner node of decision trees. This update also brings the change of evaluation of the trees. The decision tree with the *if* nodes is processed from the root node to the leaves which contains the value directly representing the decision (accept/reject). We expected the more expressive power of such trees. These trees were introduced in section 7.3.

The second major change in the sub-problem definition is the presence of the buffer for accepted tasks in a worker agent. We started our experiments with this buffer present in a worker agent. Later we tried to remove this buffer with the following idea. When an agent cannot accumulate the suitable tasks in the incoming buffer for later

task buffer removal

processing, the decision of task acceptance is harder because when a new task is accepted the processed task must be interrupted and therefore returned as unsolved. This behavior has an impact on the time criteria because a part of the task can be solved multiple times. Therefore this change introduces more responsibility to the decision making of worker agents.

Evolution as reinforcement learning problem

In contrast to the supervised learning, we have no training set in the problem of task allocation in MAS. There are no pairs of *the decision tree–its performance* available on a given task-set. We cannot say how good the tree is without a simulation. This approach is known as Reinforcement learning [3, 80]. It can be characterized as follows:

- **Input**: The input should be an initial state from which the model will start.
- **Output**: There are many possible outputs as there is a variety of solutions to a particular problem.
- **Training**: The training is based on the input. The model will return a state, and the model is scored based on its output.
- The model continues to learn or is being evolved.
- The best solution is decided based on the maximum reward.

During the simulation, the randomness of task distribution has an impact on decision tree performance. Therefore, the repeated evaluation of each tree performance is done for better estimation of the real tree performance. Unless otherwise stated, each tree evaluation is made three times. The resulting value of the fitness function is averaged over these runs.

As stated above, the evolved decision tree is common for all worker agents in the experiment. It is, of course, possible to control each worker by a different decision tree. The idea of evolving a set of decision trees at once was left beside because it brings yet another complexity to our goal. It may be investigated later as a follow-up work.

agent and task attributes

The attributes of the currently offered task and the worker agent itself are taken into account in the decision-making process. The set of attributes of the task consists of metadata of input dataset, e.g. expected suitability of the particular model for given task or number of steps needed for processing. The values of dataset suitability are estimated based on precomputed results. Not only dataset related attributes are used. Another group is bound to the worker agent, e.g. a number of processed steps of the currently computed task or a number of solved tasks. The information about actual criteria is not used for decision making. The criteria are fixed for the whole evolution in a particular experiment. Therefore the evolved trees may not be suitable for the decision-making process in the experiment with the same task set but with different criteria.

As a baseline for a result comparison, the probabilistic decision-making process is used. When the task is offered to a worker agent, it accepts the task with a given probability. Also, few decision trees were constructed by hand and compared to the evolved ones.

The automated evolution of trees by GP faces several problems which will be described later in detail (see section 6.1). The main two, which are related to each other, are bloating and destructive impact of crossover operator. The bloating is a phenomenon of unattended grows of the tree structure. It can be caused by genetic operators, both the mutation and crossover.

crossover operator challenges

The second mentioned crossover operator also has a considerable influence on the phenotype (the semantic/meaning) of the individual. It is commonly implemented as a swap of two subtrees from distinct parent trees. When such a swap is made without the great attention, the subtrees selected for the swap can have a different genotype and also the phenotype. The exchange of such subtrees disrupt the good building blocks in newly created individuals. Therefore, we focused our effort to find suitable subtrees in the crossover operator.

1.4 RELATED WORK

In its general meaning, the idea of task distribution is not new. Computers got the ability to communicate among themselves in early 60's, and it was natural to share tasks between them. The engineers had to face hardware limitation, networks were slow, and their capacity was limited, even the capacity of high-speed memory was insufficient. In that times it was inconceivable to compute tasks of nowadays scale. The boom of data mining came with the development of computers' speed and its memory size in 90's. The solid theoretical frame of task distribution came even later [5, 53].

A parallel or better to say a „distributed" data mining came into the center of attention of researchers [16, 101, 103]. The interconnections between the areas of MAS and data mining were also studied [33, 36].

This work is inspired by the work of Roman Vaculín [86] and the Bang3 project [8, 87]. Artificial intelligence models were used in adaptive agents to solve data mining tasks within a MAS. Some concepts of Belief, Desire, Intention (BDI) computational agents and methods of possible decision making of these agents were described there. The above-mentioned work [86] also deals with ontologies of agents' communication.

automated evolution of bottom-up oriented decision-making systems

In our work, we decided to focus on computational agents that solve data mining problems and automatic evolution of bottom-up oriented adaptive decision-making systems. The GP is the core principle of automated evolution. Some aspects of GP were investigated in detail. Particular attention was paid to methods of crossover operators. There exist efforts [10, 61, 85] which try to take care about the semantics of subtrees during the application of the crossover operator in GP. It is an actual topic in artificial intelligence research studied in recent years. We tried to propose some answers to the question – „How to measure a subtree similarity?" We believe that there is some space for exploration and new research.

The properties of a GP process are frequently investigated and tested on the problem of Symbolic regression [85, 102]. White at al. [94] initiated a survey on how to do better benchmarks of GP on the problem of Symbolic Regression. We use some proposals and also selected datasets mentioned there.

Other related work is mentioned when new terms or fields are introduced – typically in the parts Background and State-of-the-Art (ii) and Overview of Our Approach (iii).

1.5 THE OUTLINE OF THIS THESIS

The text of the thesis is divided into four logical parts. Following lines guide you through the individual parts of the thesis.

The first part, *Introduction*, tries to introduce the reader to the motivations and intentions of the author to dedicate several years to the research of a given field of Artificial Intelligence (AI). This part also defines the problem and subproblems of the presented thesis.

The following part *Background and state-of-the-art* is composed of four sections as follows: *adaptive algorithms*, *multi-agent systems*, *computational intelligence* and *agent control*.

The section 2 about *adaptive algorithms* brings at first a little insight to the concept of adaptivity. It discusses it also from the algorithms point of view. The following section introduces the concept of the *agent*. There are definitions of what the agent is, what are its abilities, restrictions, or properties. How can an agent interact with its environment and how it can describe its model of the world where it is situated. The BDI architecture is shortly mentioned.

We move from the individual agents to the MAS in section 3 (*multi-agent systems*). The short introduction shows that such systems can be understood from many points of view. The key characteristics of the MAS are presented concerning the degree of heterogeneity and degree of communication between agents. The role of the MAS is discussed from the developer's aspect and known standards for implementations are mentioned.

The section 4 (*Computational Intelligence*) is dedicated to a brief insight into a vast area of interest – the data mining. We focused on concepts, models, and techniques used later in our research. The methods and models which are encapsulated in our working agents are described closely. There are also many other methods which cannot be naturally covered or even mentioned by this limited section. Artificial neural networks are in several modifications encapsulated in our agents, but it is also a well known and old concept. Therefore only a brief description is presented.

Evolutionary algorithms are the essential part of this research. With a focus on the GP, the idea of utilization of search algorithms inspired by nature is described. The paradigm of artificial evolution is shown from its roots in the late '50s. Nevertheless, a deeper insight into GP is the content of a separate chapter later.

A core of our experiments is in distribution of individual tasks from the whole task set which will be described in section 4.2.

Section 5 (*Agent control*) guides the reader through the basic concepts of agent's communication languages to the more complex concept of ontologies. The agent could be controlled by an unchangeable set of rules/routines or in an adaptive manner. These two approaches are discussed in detail and also from the point of view of controlling computational agents determined to solve data mining tasks. The adaptiveness of the agent's control is dependent on time and the state of the environment and some aspects of this are mentioned. The section is concluded by a brief overview of practical problems of agents and their developers or designers.

The part iii, *overview of our approach*, is divided into two main sections. The first is dedicated to the GP. A description of methods concerning population generating opens this section. Next, the common mutation and crossover operators are mentioned, and also an adaptive approach to their application is proposed. The most used structures in GP are *trees* – well known from graph theory. The necessary part of the formalism of trees is presented here. During the application of a crossover operator, the algorithm has to deal with a bloating problem. We focused on this challenge and tried to suppress the consequences of bloating by interchanging the similar subtrees of candidate individuals. Several methods for measuring a subtree similarity are proposed and evaluated.

The second section of the part iii (*overview of our approach*) is named *experiments* and connects papers of the author published in conference proceedings or journals. Articles are ordered by the date of their publication to help the understanding of partial contributions to the goal of the thesis. The first three articles are about single & multi-criteria optimization during the evolution of decision-making systems for controlling data mining agents. The article – Matching subtrees in genetic programming crossover operator – compares six methods for measuring tree similarity. It is the answer to the fourth step of our approach – To explore the possibilities of improvement of crossover operator in GP. The topic of the following article is Tree-based decision strategies and auction systems in computational multi-agent systems. It concentrates on the process of task distribution and task solving in MAS by engaging auction systems. It compares the proposed approaches to the problem of agent control in task distribution in MAS.

The final part *conclusion and future work* summarizes the contribution of the author's work. Some ideas for further research are also presented in chapter *future work*.

Part II

BACKGROUND AND STATE-OF-THE-ART

2

ADAPTIVE ALGORITHMS

This chapter is focused on a concept of adaptivity. The idea of adaptive algorithms will be discussed.

2.1 ADAPTIVITY

Adaptivity or adaptiveness is the state or quality of being adaptive; to have the capacity to adapt. The adaptive entity has to be able to modify itself or its parameters to be capable of reacting to the changes in the environment [96, 98].

adaptivity is well known from the evolution of species of animals or plants

The term adaptivity is well known from the evolution of species of animals or plants. When the environment changes and the new conditions are not acceptable for the species, they have few options: change environment (move somewhere else or affect current environment), adapt or die. The first option – to go to another environment – has a fast effect and needs small or no changes to the object itself; however, sometimes it is not possible. To modify current environment is an effective solution but often the object is too small or has too weak influence that the changes in the present environment are not sufficient. Death is one from the two remaining options to choose from for species which are not capable of changing the environment. Death is commonly not a preferred solution. The last chance is to adapt itself.

The common examples of adaptivity are everywhere around us. The dog changes its hair twice per year – thick fur for the winter times and thin for warmer part of the year. Some of the ancient mammoths, after the end of ice age, lost their hair and diminished and during the thousands of years developed new species – nowadays we know them as elephants. From the kingdom of plants, we know hundreds of varieties of apples. Apple trees are growing in all corners of temperate zones in many different local conditions. As another example, we can look at cactuses – the plants capable of surviving many years in the desert environment. They omit their leaves to minimize the surface from which water can evaporate. They store water gained during the occasional rains in their pulp. That is a perfect example of adaptation.

2.2 ADAPTIVITY AND ALGORITHMS

ability to modify parameters of the program from program itself

In the domain of computer programs or algorithms, the adaptivity lies in the ability to modify parameters of the program itself during the program run. An adaptive program has to be capable of modifying and tuning its internal parameters as a reaction to the changes in the environment where it is situated. Its behavior depends not only on input data but also on the current state of the program run, environment variables, etc. Modern interactive computer programs are often adaptive – at least they commonly provide more than one language for its Graphical User Interface (GUI). It is very comfortable when such program changes the interface language automatically based on the language of user's operating system. In other words, it adapts itself to the environment of the Operating System (OS).

3

MULTI-AGENT SYSTEMS

Interest in multi-agent systems grew in late 90's [58, 91, 93, 97] but the original ideas came from Object Oriented Programming (OOP) whose roots lead to the 60's to artificial intelligence group at MIT. The application in wide industry areas such as medicine, commerce, entertainment, or simulation, etc. [98] has shown that MAS technologies are well suited to engineering complex software systems.

In recent years, the concept of MAS has come into the interest of many research groups. There are thousands of projects combining a MAS with many other fields of AI such as *game theory*, *simulations*, *planning* or *datamining*. The *AGENTFLY* [72, 73] or the *AgentScout2* [48] projects are a good example of such symbiotic combination.

3.1 A SINGLE AGENT

As an answer to the question "What is an agent?" we can use the definition from [93]:

a software agent is, in fact, a piece of code

Definition 1 *An agent is a computer system that is situated in* some environment *and that is capable of* autonomous action *in this environment in order to meet its design objectives.*

We can imagine a software agent as a piece of code, a small snippet or practically a whole package/module which can communicate through defined Application Programming Interface (API) via some defined system of messages. Such an agent usually has some goals for which it was designed. It can be almost anything from simple tasks like periodical logging of the temperature from sensors to complicated crawling of websites for semantic analysis or high-performance automatic stock trading (also known as algorithmic trading).

environment state

What is the *environment*? Normally it is a runtime environment of given programs extended by some program API. For example: we can imagine the environment for some instance of the Java class as some memory block for the heap, some stack and program counter in JVM. However, this case can interact with some other environments such as a web server through TCP/IP socket or file system through native OS functions. For the physical robot, the environment is its real neighborhood which he can perceive and to which he can influence.

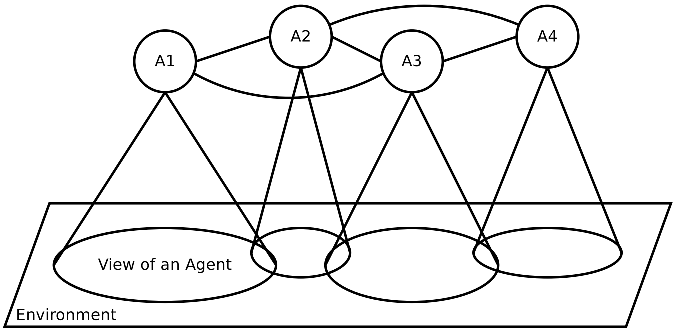


Figure 2: Multi-agent system: Agents with their relationships and intersecting areas of influence of Agents in an environment.

Formally speaking [47]:

Definition 2 *Agent's environment can be characterized by an environment state* s *from a set of possible states* S*. At any moment, the environment is in one of such states. The set of possible* actions *of the agent affecting the environment is modeled by a set* A*. An agent can be interpreted as a function which maps a sequence of environment states to an action:*

$$action : S^* \to A$$

This is a model of a standard agent.

The non-deterministic response of the environment to the agent's action is: $env : S \times A \to 2^S$. If the set $env(s, a)$, which is a result of the application of an action a on a state s, is a singleton for all such combinations of states and actions, the environment is *deterministic*.

autonomous agent can perform actions without the intervention of human or other system

Autonomy is used to express the fact that agents can act (to perform actions) without the intervention of humans or another system. Agents' important features are adaptivity to changes in the environment and collaboration with other agents. Interacting agents join in more complex societies – *multi-agent systems*. These groups of agents gain several advantages as the applications in distributed systems, delegacy of subproblems on other agents, and flexibility of the software system engineering.

An *intelligent agent* is one that is capable of *flexible* autonomous action in order to meet its design objectives, where flexibility means three things: *pro-activeness* (goal-directed behavior), *reactivity* (response to changes), and *social ability* (interaction with other agents). These properties can be understood as requirements to an intelligent agent.

planning vs. reactive agent

We can classify agents from several points of view. The first one is by their architecture in the meaning of ability to plan their actions vs. pure reactivity. The simplest example of a pure reactive agent is a thermostat. When it is cold, it starts to heat, when it is warm it starts to ventilate. In contrast to the thermostat, we can imagine an AI player in some strategic game. To win it needs to precisely prepare plans which consist of many dependent sub-plans like: get enough amount of resources, build a factory, produce exploring units, locate the enemy, produce war units and destroy the enemy base. For reactive agents there is one very important requirement – they should act faster than the time of shortest change of the environment. Here we suppose no delay from sensors to decision unit of the agent. This is a stringent requirement to agents' designers and they meet it hardly ever.

interaction between agents

The second point of view in classifying of agents is by the amount and type of mutual interaction. There are agents with no interaction with others. For example, several robots have to crawl the websites to mine some information. They do not have to communicate with each other. In contrast to not communicating a set of agents, we have many examples of interacting (i.e. communicating) groups of agents. It says nothing about concurrency or that agents cooperate in one team.

These groups of agents can communicate in order to achieve some team goal – then we talk about cooperating agents. On the other hand, they can communicate only for the gain of some information which helps them to beat their opponents. Often we cannot distinguish between those two types of multi-agent systems. As an example of the combined system, we can choose some computer real-time strategy game with AI. When we play against several opponents controlled by AI sometimes, they enter into alliances to gain more power and influence over the game's world. However it depends on AI if alliances remain to the end of the game or whether they change them (and how many times).

There are too many aspects how we can understand problems of the classification of agents. Specific types of agency are described in chapter 4 of *Readings in Agents* [40]. From a general perspective, the agents and MAS are a broad discipline. Thus it seems to be useful to bring some "formal" descriptions of agent's architecture. A closer view of MAS taxonomy will be presented in section 3.2.

3.1.1 *BDI Architecture*

The BDI architecture is the one which views the system as a rational agent having certain *mental attitudes* of Belief, Desire and Intention, representing, respectively the information, motivational, and deliberative states of the agent [67].

The structure of BDI architecture is shown in figure 3. It is a model of knowledge base system with three sets of knowledge which are manipulated by different parts of an agent control.

Believes represent agent's knowledge about an environment – a model of a world – and some other facts like agent's internal properties and

[1] Based on the diagram from BORDINI, Rafael. Programming Multi-Agent Systems in AgentSpeak Using Jason.: `http://www.dur.ac.uk/p.h.shaw/teaching/mas/lectures/rafael/MAS-Lecture05-6up.pdf`

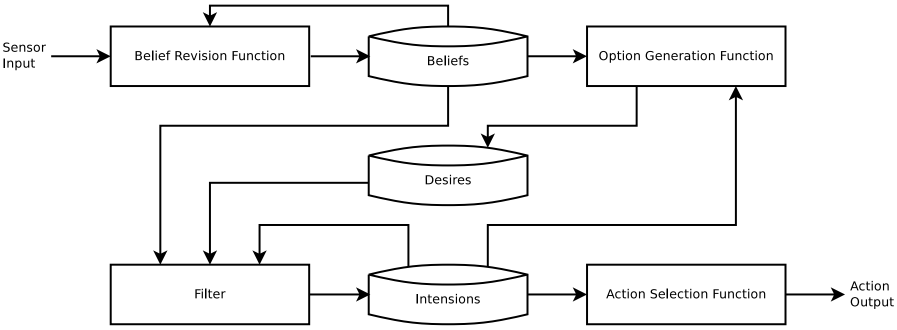


Figure 3: Simple Belief-Desire-Intention agent architecture[1].

states. This set is often represented symbolically. It is important to say that beliefs can be incorrect and can be changed during agent's life.

the BDI agent wants to achieve its desires by performing actions

Desires are agent's wishes such as desired states of the world. The agent wants to achieve its wishes. Some of the desires may not be achievable or one can rule out another. We can divide desires by time horizon to "short-term" and "long-term" – it is very relative what does "short-term" or "long-term" mean.

Intentions represent agents actions, which the agent can choose to fulfill its goals/desires. It is possible that one intention is the same for several agents.

3.2 MULTI-AGENT SYSTEMS

Several taxonomies of multi-agent systems have been presented previously for the related field of distributed AI [79]. For example, Parunak [64] has presented taxonomy from an application perspective. The essential characteristics of MAS are:

- system function (designed functionality)
- agent architecture (degree of heterogeneity, reactive vs. deliberative)
- system architecture (communication, protocols, human involvement)

Stone [79] preferred to observe MAS taxonomy from two most important aspects of agents – degree of heterogeneity and degree of communication. He presented the following overview of such classification – see the table 1.

This work focuses on the MAS with heterogeneous communicating agents. In such MAS, the agents interact with each other according to specific protocols. We can adopt the definition of MAS from [24]:

Definition 3 *A MAS is a loosely coupled network of agents that work together to solve problems that are beyond the individual capabilities or knowledge of each agent.*

Table 1: Issues arising in the various agent's scenarios as reflected in the literature

Homogeneous Non-communicating Agents • Reactive vs. deliberative agents • Local or global perspective • Modeling of other agents' states • How to affect others	**Heterogeneous Non-communicating Agents** • Benevolence vs. competitiveness • Stable vs. evolving agents (arms race, credit assignment) • Modeling of others' goals, actions, and knowledge • Resource management (interdependent actions) • Social conventions • Roles
Homogeneous Communicating Agents • Distributed sensing • Communication content	**Heterogeneous Communicating Agents** • Understanding each other • Planning communicative acts • Benevolence vs. competitiveness • Negotiation • Resource management (schedule coordination) • Commitment/decommitment • Collaborative localisation • Changing shape and size

According to [44], the characteristics of MAS are:

- each agent has a limited viewpoint – it has an incomplete information or capabilities to solve the problem;
- there is no global system of control;
- data is decentralized;
- computation is asynchronous.

datamining computational methods are encapsulated in the agent

In the context of a MAS implementation of data mining system, the computational methods are encapsulated in a computational agent who has a comprehensive set of parameters. Such computational agent accepts datasets and performs some appropriate processing of them. For complex MAS, where the society and states of agents are changing dynamically, the problem arises to find an appropriate provider solving some subtask for the requester.

For the designers and developers of complex MAS, it comes in handy to observe the system as a sociological concept of *roles*. The role is an expected behavior of all agents playing given role. It consists of a set of action patterns, rights or responsibilities. An agent can enter such system only in the case that it accepts and plays one of the available roles in that system. Each part of the system can be designed separately – communication protocols, computational routines, or environment interaction routines – thanks to the knowledge of the particular role of a given entity.

There are several platforms and standards regarding the implementation of multi-agent systems and agent interaction e.g. the BDI-based logical language AgentSpeak [68], or the Knowledge Query and Manipulation Language (KQML) [31]. The newer FIPA-ACL [32] specification (Agent Communication Language) and its Java implementation, called Java Agent Development Framework (JADE) [12] are the leading and the most extensively used ones.

experiments were performed on top of JADE framework

For our experiments, we utilized the JADE as a platform for all the agents and their environment. It is an open source platform for peer-to-peer agent based applications. It can be distributed across machines which do not even need to share the same operating system. A set of GUI tools for debugging and control is provided.

4

COMPUTATIONAL INTELLIGENCE

When we talk about computational intelligence, we have in mind nature-inspired approaches which are suitable for solving problems where traditional methodologies are ineffective or infeasible. First principle and probabilistic or *black-box* solutions belong to traditional approaches [26, 28].

In the computational intelligence, it is also important to take care not only of methods and models but also of the proper use of them. Each specific data needs a specific approach. The selection of suitable models and algorithms can be made with the utilization of *metadata* – the information gathered from data itself. The *meta learning* is a subfield of Machine learning where automatic learning algorithms are applied on meta-data about machine learning experiments [15].

4.1 DATA MINING

Data mining, also known as Knowledge Discovery in Data (KDD), is a relatively young interdisciplinary field in computer science. Data mining techniques have been widely applied to problems in industry, science, engineering, and government, and it is widely believed that data mining will have a profound impact on our society. The growing consensus that data mining can bring real values has led to an explosion in demand for novel data mining technologies [17].

The overall goal of the data mining process is to extract knowledge from an existing data sets and transform it into a human-understandable structure for further use [17]. This process usually consists of following steps: model selection, preprocessing of data, modeling the data, model validation, postprocessing, and visualization. In particular cases, some blocks of this process can be omitted.

datamining process tries to obtain knowledge from existing data in human-understandable structure

The data-mining tasks can be divided into several categories. The *regression* is the task to find a function which maps input to output values. It estimates the relation between input and output. The performance of a regression model can be evaluated by Mean Squared Error (MSE).

A *classification* is an approach how to assign a class to a new instance of data based on class of the most similar data from a training set. It is a special case of regression where the mapping function has a categorical output value. Binary classification is a case where only two target values (classes) are present in data. For the binary classifiers, there is the *precision* (positive predictive value) and *recall* (sensitivity) as measures for evaluating the quality of the binary classification.

The *clustering* comes into account when we need to group a data based on their similarity according to implicit laws in these data. It is a case of unsupervised learning [18].

There are also algorithms from the family of *Instance based learning* [70]. These algorithms are based on a comparison of an unknown instance of data to those seen before. This is in contrast with previously mentioned groups of methods where the explicit model of data is created.

training and testing data

The main goal of data-mining is to model true relations in given data and not to model observations on these data. The learning of false relations and characteristic leads to *over fitting*. To avoid this undesirable phenomenon, we need to split data into two disjoint sets – *training* and a *testing* datasets. On the training data a model is constructed and tuned, and on the testing data, the model is evaluated. The popular approach to this problem is a *K-fold cross validation* [18] – the whole dataset is divided into training and testing subsets K times (commonly ten times) and in each run, the split points are moved. This method is robust and for example, it deals well with sorted data.

For the preservation of clear and well readable text in this thesis we tried to stay on the specific level of details description, so only important and closely related things used in our work will be mentioned.

Notions

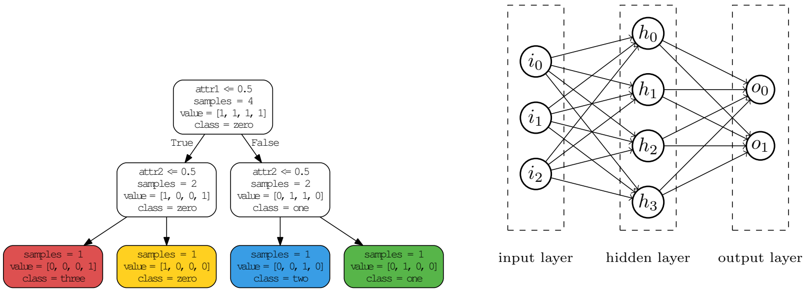
DATASET is some subset of the processed database. We often split the dataset into learning and testing parts.

DATA PREPROCESSING is the first important step in KDD process whose main goal is to filter out invalid, extreme or other unsuitable items from the given dataset; or to cope with missing values.

MODEL is some statistical formalization of relationships between variables.

DATA MINING PROCESS commonly consists of three basic phases – selection and preprocessing, building and verification of model and results interpretation.

CLUSTERING is a task of discovering groups and structures in the data that are in some way "similar", without using known structures in the data [84].



(a) Decision tree (b) Multilayer perceptron

Figure 4: Example of data mining models[1].

CLASSIFICATION is learning a function that maps (classifies) a data item into one of several predefined classes [84]. For example, an e-mail program might attempt to classify an e-mail as "legitimate" or as "spam".

REGRESSION attempts to find a function that models the data with the least error [84].

DECISION MAKING is a task where some suitable model of relations between data is used to help some entity (either program or human) to make a right decision.

4.1.1 *Methods and models overview*

Following section will contain an overview of commonly used KDD methods and datamining models. This list is only a concise review of concepts from data mining domain used in our work. There are also dozens of statistical methods. The mentioned terms can be better specified and divided into more precise categories. Those which we use in our work will be later described in closer detail.

NAIVE BAYES is one of the classifiers, very simple and applying Bayes′ theorem [90]. It is a probabilistic model.

SUPPORT VECTOR MACHINES are based on the idea to create linear classifier so as to have the gap between the separation hyperplane and each category as broad as possible. The classification of the linearly non-separable data is done by a transformation into high-dimensional feature space with so-called kernel trick [47].

K–NEAREST NEIGHBORS chooses for the new data instance the k most similar training instances. The class with the majority votes of the neighbors is assigned [47].

DECISION TREE uses a tree-like model to classify some data. An example of a simple decision tree is in the figure 11.

RADIAL BASIS FUNCTION is a function whose value depends only on the distance from the origin. Commonly it is a distance function and the norm is Euclidean distance but others are possible too. Radial Basis Function (RBF) is used as an alternative activation function in feedforward neural networks.

MULTILAYER PERCEPTRON is a feedforward artificial neural network model. It maps sets of input data onto a set of appropriate output. It has several layers of nodes (neurons), between the nodes there are oriented edges and the lower level is fully connected to the higher – only input may not be fully connected. Each neuron has some nonlinear activation function. MLP is learned by supervised learning method called backpropagation. An example is in figure 4.4(b).

4.1.2 *Neural networks*

Artificial Neural Networks (ANN) are inspired by structure and function of biological neural networks in human or animal brain. The primary element of ANN is artificial neuron often called a *perceptron* – we can see it in figure 5. It sums n weighted inputs and passes the sum to some activation function ξ. A sigmoid or Heaviside step function is often used as activation functions. When the value of this function overcomes some threshold Θ then neuron "fires" signal to output z.

The neural network has several layers of nodes (neurons) – see fig. 6. Between nodes, there are oriented edges and lower a level is connected to a higher one (input to hidden and hidden to output). A minimal number of levels is two – input layer and an output layer. However, this is not enough and we need some so called hidden layers. Thanks to many connections, hidden layers, and complicated structure, it is tough to explain how a trained ANN solves a given task – it is usually a black box for us.

Commonly we first train an ANN on a training dataset and then we give new data to the neural network and it recalls learned associations and gives us the appropriate output. There are many algorithms how to learn neural network which can be divided into three classes: supervised or unsupervised learning and reinforcement learning.

[1] Sources of images: `http://gautam.lis.illinois.edu/monkmiddleware/public/analytics/decisiontree.html` and `http://www.dtreg.com/mlfn.htm`

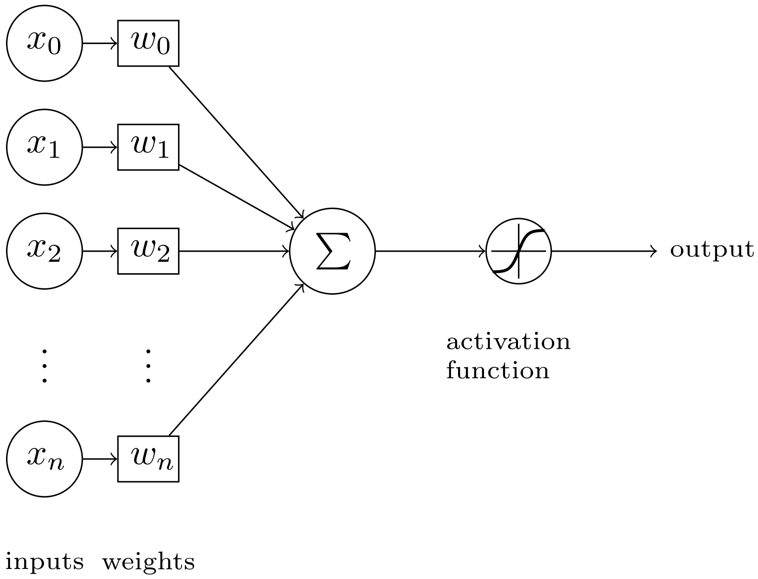


Figure 5: Simple perceptron model with n inputs.

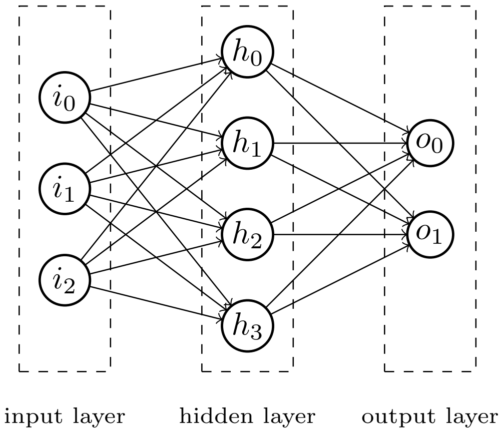


Figure 6: Example of a Multi layer perceptron. An example has three input perceptrons ($i_0, \ldots, i_2$), four neurons in a hidden layer and two output neurons o_0, o_1.

4.1.3 *Evolutionary algorithms*

The roots of Evolutionary Algorithms (EA) lead to 50's – 60's. EAs have several independent sources; the first pioneers were Fraser, Bremermann, and Reed, next Fogel, Holland [38, 39], Goldberg [14, 34, 35] or Koza [50–52] in the late 80's. The EAs are a subset of generic population-based optimization algorithms. They are inspired by biological evolution – selection, recombination, mutation, and reproduction.

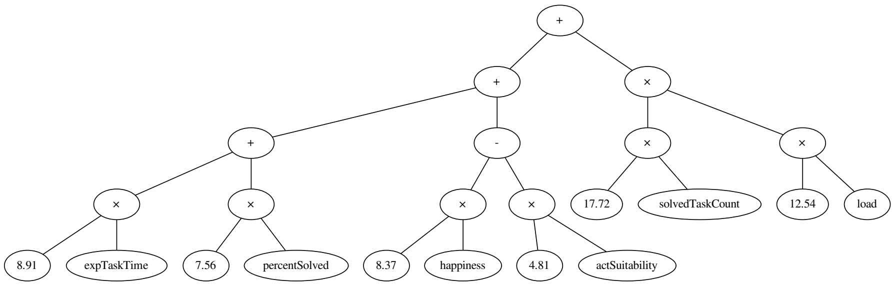


Figure 7: An example of a tree which represents a polynomial

genetic algorithm

The well known and commonly used example of EA is the genetic algorithm. We have a set of encoded solutions and by applying operators like a mutation and crossover, we explore the state space of a given problem and look for a better solution in the neighborhood of the current solution. The quality of the solution is measured by a *fitness* function.

There are many other approaches like an evolutionary programming, differential evolution, neuroevolution, etc. Moreover, of course – the genetic programming which we will make an extensive use in this thesis.

4.1.4 *Genetic programming*

genetic programming stands on the evolution of tree structures

Solutions produced by genetic programming are computer programs or evaluable formulae whose fitness is determined by evaluation of this programs. Traditionally the programs are encoded as a tree structure. The evaluation of the tree is relatively straightforward thanks to the utilization of recursion.

In the inner nodes of the tree there are operators and in the leaf nodes, we find terminals. The genetic programming is ideal for languages like Lisp or other functional programming languages. On the other side, it is possible to use not tree-like structures for evolution, e. g.linear programs in imperative programming languages or even grammar which generates valid codes. There are also frameworks as JGAP or GPC++, which help developer to implement fast and simple genetic programming experiment. On figure 7 we can see an example of GP evolved tree. This tree represents six properties connected in one polynomial by operators $+$, $-$ and $\times$. The genetic programming is one of core topics of our work and will be described in detail in section 6.1.

4.2 TASK DISTRIBUTION

When a MAS is designed to solve a large data-mining problem, the idea of dividing a problem into smaller pieces (subproblems) comes into the foreground. The natural way is to distribute tasks to the solving units (computational agents) and collect the results. The question is: How to distribute the tasks in order to meet the designed objectives?

The task allocation in MAS is a part of multi-agent planning techniques. The wide overview of multi-agent planning techniques is in [92], for the survey on *cooperative* planning see [83]. In [92], there are five phases of solving multi-agent planning problem distinguished. The first one is described as *allocate goals to agents*. The others are named: refine goals to subtasks, schedule subtasks by adding resource allocation, communicate planning choices to recognize and resolve conflicts. All these phases can and often interleave each other. Applied to our problem, the allocation means – which agent commit to solve which task instance. Our agents are not cooperative – they communicate not to refine their plans or allocation of resources. The communication between workers and manager is only for exchange of task instances and the results of processing.

It is offering to take some meta-data of the particular dataset into account. It is also important to have some additional knowledge about the used data-mining models – a model's metadata. A proper connection of these two sets of knowledge is essential to building a well-performing task distribution.

bottom-up oriented distribution of tasks

Another approach is to distribute the tasks randomly – it is a good baseline for comparison of the performance of other methods. In our case – it is a way how to transfer responsibility for matching task–model from a central authority to the solving entity. We drop this responsibility from our *manager agent* to the *working agents*. After the delivery of the offered task, they should make a decision whether they accept or reject that task. A detailed description of tasks distribution is presented in experiments – see the chapter 7.

5

AGENT CONTROL

Each agent has some mechanism which guides its steps. For a software agent, it is some method, or it may also be a whole complex system consisting of many parts. It may be *static* for the whole agent's life or it may be adaptive and variable in different states of the agent's life.

5.1 LANGUAGES FOR AGENTS

„By an *agent* language, we mean a system that allows one to program hardware or software computer systems regarding some of the concepts developed by agent theorists“ – Wooldridge [99]. The requirements on such language are not strictly defined. At least, the language should provide constructs which correspond to purpose and capability of the agent. There could also be present some attributes of agency mental models (a.g. beliefs, goals). The *AGENT0* language was 0 prototype for illustration of principles of agent oriented programming. Later reimplemented by Thomas [82] in her doctoral thesis as a *PLACA* – Planning Communicating Agents in 1993. There are also many newer languages e.g. *Jazzyk* [62] but a lot of them are only prototypes serving for experimental purposes of their authors. For a complex overview of agent languages theory see section 4 in [99].

5.2 ONTOLOGIES

ontology formalizes agent's understanding of its environment

An ontology is a formal, explicit specification of the terms in the domain and relations among them [63]. Terms are called *concepts* and are arranged by relations into a hierarchy. The set of relations between concepts is a terminology in our domain. The knowledge of an agent consists of terminology and instances of concepts. An example of ontological language is Web Ontology Language (OWL)[11].

Figure 8 depicts the ontology of family relationships [86]. For instance, the *Father* concept is defined as a subclass the *Man* and the *Parent* concept. The *Parent* concept is defined as a *Person*, whose child is also a *Person*. Having these concepts defined we can describe individuals. For instance, the individual *MAGDA* is an instance of the *Mother* concept and *PAVEL* is a child of *MAGDA*.

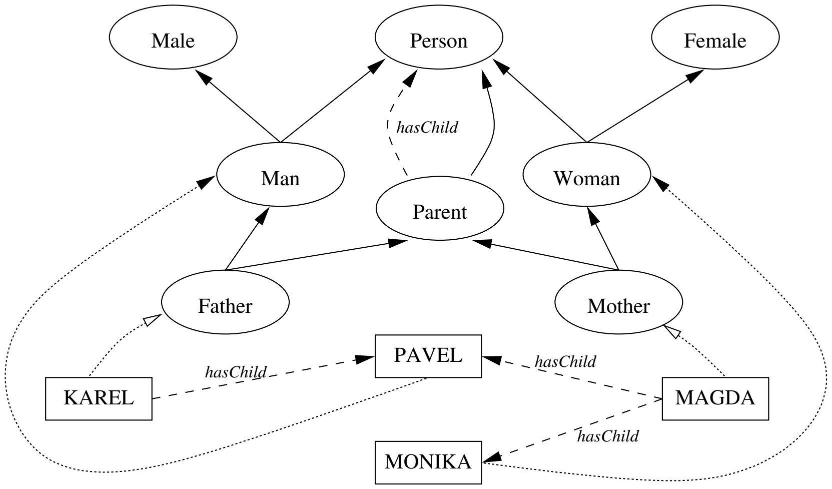


Figure 8: Ontology of family relationships[1].

As stated in [63] the ontologies bring following benefits:

- share common understanding among people or software agents,
- reuse domain knowledge,
- make domain assumptions explicit,
- separate domain knowledge from the operational knowledge,
- reason about concepts.

5.3 HARD CODED CONTROL

hardcoded agent is not capable to change its behavior

The hard coded control means that the agent has its „brain" designed by its author and implemented for example as a static set of decision rules. This agent has no possibility to extend its set – it cannot learn or develop new rules to achieve new abilities. Such agent can only extend its knowledge about the environment, e.g. actualise the map of nearest space around itself.

There are many methods how a designer can encode agent's behavior. From elementary and straightforward *if–else* or *switch–case* approach suitable for very simple behaviors like control of an air-conditioner, to the formal systems for a description of knowledge: modal logic, Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). For an introduction to the formal reasoning we can suggest work from Michael Huth and Mark Ryan [41].

CTL allows us to model system (e.g. agent's behavior) in time like a tree structure. From the current point in time, there are different

[1] Source of the picture and its description is [86].

paths to several points in future. We can claim things like: „An agent *A* can jump in all circumstances from now forever.“, formally:

$$\mathsf{A}\Phi$$

where A is quantifier with meaning „along All paths“ and Φ is a predicate saying „An agent is able to jump.“ Another example:

$$\mathsf{E}\Phi$$

where the E quantifier means possibility (along with at least one path). We can read it as: „In some cases agent *A* is (or will be) able to jump.“

5.4 ADAPTIVE CONTROL

Many problems can be solved by the agent(s) with hard coded (static) reasoning system. Especially the problems for which the optimal or at least sub-optimal solution is known.

There are of course many interesting, actual or difficult problems for which humanity does not know a reasonable solution. Our daily lives depend on many complex systems like public transport, water and electricity supplement, goods distribution and many others. These systems are controlled by sophisticated largely unmanned computer systems. These life-critical systems are developed by tens of experts for the given domain. They must also be precisely tested and monitored. Thus developers often use traditional technologies and approaches [45, 49].

complex variable systems need adaptive approach

Such traditional approaches may not be optimal and solve given task „only“ so good as it is safe enough. However, better (not necessarily optimal) solution may save many resources. Every saved kilometer or five minutes in large logistics company with hundreds of trucks means lower costs and advantage against rivals. However, the environment in which logistics company operates is changing very fast – traffic jams, detours, changes at suppliers, etc. So the control and planning software system should be adaptive and should deal with these problems. In the optimal case, it should be able to learn from current and past states and continuously improve its deliberation and planning.

Adaptive control can be done by many approaches and depends on the designer and selected architecture and also on the current implementation. From the global point of view, we can divide adaptive control into two classes – the first one contains such control systems which are adaptive continuously during the whole agent's live (system run). The second class associates control methods whose adaptivity means that there exist some attributes which are changing. These attributes are incorporated into the agent's decision-making process.

parametrization of decision-making systems

Let us have some decision-making system such as a neural network, a decision tree or associative rules. All these systems can be parameterized – in neural networks, there is variability in topology: the number of hidden layers, connections between layers, or parameters like a learning rate or momentum factor. With decision trees, we can parameterize maximal depth or number of branches leading from one node. When the expression in node has arity two – eg. *if* $(X \geqslant 42)$ *then {. . . } else {. . . }*, the number of branches is two. Greater arity such as *if* $(X \geqslant 0 \wedge X \leqslant 42)$ *then {. . . } else-if* $(X > 42 \wedge X \leqslant 666)$ *{. . . } else {. . . }* leads to three (or more) branches from a given node. Of course, the set of possible functions in nodes is also a parameter of a decision tree system.

There also may be some other parameters representing the state of an agent or agent's model of the environment. All of the above mentioned parameters are subject to change during agent's live/run. Thus agent's reasoning may change during its life. An example is presented below:

We have a small cleaning robot with two cameras. The first camera is used for a visible spectrum of light and the second one for infrared spectrum. The robot uses output captured from these cameras for exploring the environment, to avoid obstacles and for navigation. The robot also has a light level sensor, which measures the amount of surrounding light. We leave it (powered on) at home during the evening. When the sun goes down, the robot recognizes a bad light condition (*a parameter was changed*) and switches the active camera from normal to infrared.

There is one general case in which the adaptive control is suitable. Sometimes designers do not know how to propose the control algorithm. Merely because they do not know the optimal form of a solution. Very often they even do not know anything about „good" or at least „suitable" solution. They have to use advanced methods to explore the space of solutions and find any solution which will be good enough.

A small recapitulation of agent's reasoning classification follows: We have several possible cases:

1. agent's control is „hard coded" – agent is not able to change its behavior during its life,

2. agent's control is adaptive

    a) the agent can learn and improve its behavior,

    b) the agent cannot learn, but its current behavior depends on the state of its attributes,

    c) agent's control is a combination of the previous cases (2a) and (2b),

    d) alternatively it is designed/found by some advanced method – not straightforward by the developer.

This case when the developer does not know „good" or at least „suitable" solution and is fascinating and makes the important part of this research. We focused on GP to help us to find a proper decision-making system for data mining agents (2d). Our agents cannot learn (2b) but change their behavior based on their attributes.

5.5 CONTROL OF A COMPUTATIONAL AGENT

A computational agent is designed to maintain several important parts of automatic data mining process. This process begins with obtaining the raw data. Acquired data need to be preprocessed. Incomplete rows of the dataset should be thrown away or completed by properly estimated values. Frequently it is also necessary to scale the values to some proper range (typically $0 - 1$) as some models and algorithms have specific needs on their input data. The outputs of *preprocessing* are the clean, normalized data suitable as input for the following steps.

The second step for our computational agent is to select proper attributes of the given data to achieve the best result in the modeling of the data. The goal is to select relatively small set of highly predictive attributes. A popular approach is *principal component analysis*. It is a statistical technique that can reduce the dimensionality of the data as a by-product of transforming the original attribute space. This principal component analysis and several other attribute selection methods are compared in [37].

Another option is to select a base set of attributes manually. The selected set of attributes is further available as building blocks for methods of the automatic evolution of decision-making systems. The responsibility for selecting only well-performing attributes is transferred to the evolution technique. We have inclined to this approach.

When suitable attributes are selected, the model can be created based on training data. For the remembrance of training and testing data subsets see the chapter 4. The last logical step is to validate the trained model on the testing data. Training and testing of a model can be managed in one solid block when the cross-validation technique is used.

Our experimental environment has two main types of computational agents. The first role is called a *manager* and is responsible for task distribution. For us, a task consists of data itself and metadata describing the dataset. The metadata contains, for example, information about a domain of the data – the dimensionality of the dataset, how many classes the dataset contains or data value ranges. The second type is a *worker* agent. Both of the types of agents are controlled by a set of *behaviors*.

Behavior type can be an *one shot* which means that the behavior is executed only once. More frequently used is the second category –

cyclic behavior which – when it is registered with a particular agent – is repeatedly executed until it is removed from the agent by an external event or by behavior itself.

Each part of the process described above is implemented as one of behaviors. There are also several other activities which agent has to manage such as task distribution, collecting finished tasks or aggregating the results. Specific details of agent's behaviors are described in the section 7 for every experiment.

5.6 TIME & ENVIRONMENT ADAPTIVE CONTROL

As stated above, the agent can perceive the environment and make decisions concerning the flow of time. When some action A is suitable at the time t_i, it does not have to be adequate in the time t_j. For example, when agents are playing the game of repeated Prisoner's dilemma [69], there is a strategy known as *tit-for-tat*. It is a simple strategy: when opponent betray you, in the next round you will betray him. This concept requires *memory* for storing the last agent's decision. Such memory could be limited by design – in the case of implementation of the tit-for-tat strategy, it is sufficient to remember (store) only the one last state. When our agent plays e. g. the artificial opponent in a strategy game with the *fog-of-war* feature, it will be useful to remember previous environment state even on the game fields actually covered by fog-of-war. On the other side, the presence of agent's memory is not the necessary premise to make the agent adaptive in the time.

The environment adaptivity of an agent requires agent's ability to perceive a state of an environment. The second required ability of an agent is to change its behavior based on its model of perceived environment. For example, the agent is used as the provider of service „load picture". This service needs to be parametrized by the name of the picture file. The given parameter is a part of agent's environment. When the agent recognizes the extension or directly the type of a requested image, it selects a suitable internal method (algorithm) to load the picture. It can be stated that such agent is environment adaptive.

In principle, our control methods can be understood as reactive from the strict agent's point of view. An agent, utilizing evolved tree as decision-making system, takes properties of itself and the offered task. Then it uses them in the decision-making process and performs a selected action as a consequence. It is not an issue due to the fact, that the ability to react properly is reflected into the decision-making system during its evolution.

5.7 PRACTICAL PROBLEMS OF AGENTS AND THEIR DEVELOPERS

In section 5.5 we have mentioned some theoretical problems of design of data mining agent's control. There are also some other problems like:

- weak penetration of standards,
- the complexity of environments and tasks,
- difficult decomposition of problems,
- difficult synthesis of sub results,
- incomplete information about the environment.

The domain of multi-agent systems is relatively young. Thus there was not enough time for creation and penetration of standards. The flagship is the FIPA organization (The Foundation for Intelligent Physical Agents) that promotes agent-based technology and the interoperability of its standards with other technologies [29].

The complexity of environments and tasks was outlined in section 5. Let us look at the difficult decomposition of problems. Some tasks have strong dependencies in data or data-manipulation process thus it is difficult or impossible to apply wide parallel computation. When we manage to decompose some problem, it is usually relatively „simple" to compose the sub results back to the final solution. However, in reverse tasks such as a *parallel mapping* (exploring) of some area – it may not be trivial to combine contributions from all agents together. Each of the map maker agents may have a little different light and air condition. Thus the captured photos are slightly different in color hue or orientation. The agent which combines photos to the final orthophoto map has to deal with these differences – it must find correct orientation and do some color corrections.

Part III

OVERVIEW OF OUR APPROACH

6

GENETIC PROGRAMMING

This chapter informs the reader about the core part of our work. For those who prefer the global view to our work, it may be advisable to read the section 8.1 first.

6.1 GENETIC PROGRAMMING

A genetic programming is an approach from a family of evolutionary algorithms where individual's genotype is represented by a graph, more specifically by a tree. The semantic meaning of individual's genotype is known as a phenotype in evolutionary terminology. These trees can be used for the wide area of applications. In this work, we use them for decision making whether to accept or reject an offered task. Section 6.2 is dedicated to tree structures.

genotype, phenotype

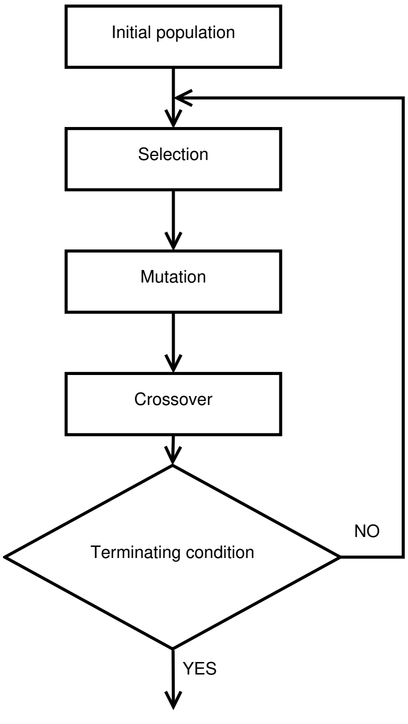


Figure 9: Common evolutionary algorithm scheme.

Common GP algorithm is described in [51]. The schematic diagram is shown in the Fig. 9. At first, the population of individuals must be created. The following step is to select individuals who proceed to the next generation. This step is omitted in the first generation. The algorithm next applies the genetic operators – Mutation and Crossover. In the case of Crossover, it is applied to a selected pair (rarely a tuple) of individuals. The final step of one run is to check whether the terminating condition is met or not.

fitness function

A crucial part of GP is the *fitness function*. It is an objective function which tells us how close is the individual (candidate solution) to the designed (targeted) optimum. The fitness function f is a relation as follows:

$$f : I \mapsto \mathbb{R}$$

where I is a valid individual – in our case represented by a tree. It maps the candidate solution to the real number. In all experiments in this work, we maximize the fitness value. An accurate description of fitness function computation will be described in the chapter 7 for each experiment separately.

6.1.1 *Methods of population generating*

initialization of trees

There are two simple approaches how to generate a tree. The first is known as *grow* [65]. This method needs parameter *max_depth* which limits the maximal depth of the created tree. The grow method checks if the maximal depth is reached – then it takes a node only from the terminal nodes. In the other case, it takes in each step a random node from a set of non-terminal or terminal nodes. If this node is taken from non-terminals then concerning its arity its child nodes are recursively created. Generating ends when all non-terminal nodes have all child nodes created. This method is illustrated by following pseudo code (Listing 3):

Listing 3: A pseudoceode of *Grow* method for tree generating

```
grow(max_depth):
  nodes = nonterminals ∪ terminals
  if max_depth <= 1:
    root ← terminals[rnd()]
  else:
    root ← nodes[rnd()]
  if has_unassigned_childs(root):
    for i in root.arity:
      root.childs[i] ← grow(max_depth-1)
  return root
```

The second commonly used method is known as *full* [65] – it distincts from *grow* method only when it decides about the set from which new node is taken. The *full* method takes nodes on all depths except the deepest one only from the set of non-terminal nodes (see line 5 in Listing 4). This method is illustrated by following pseudo code:

Listing 4: A pseudoceode of *Full* method for tree generating

```
1 grow(max_depth):
2   if max_depth <= 1:
3     root ← terminals[rnd()]
4   else:
5     root ← nonterminals[rnd()]
6   if has_unassigned_childs(root):
7     for i in root.arity:
8       root.childs[i] ← grow(max_depth-1)
9   return root
```

There is also a popular combination of this method – *ramped-half-and-half*, proposed in [51]. The idea behind this method is in the fact that the half of population is initialized by *grow* method and second part by the *full* method.

6.1.2 *Mutation operator*

The purpose of mutation is to bring – by small changes in genotype – some effect on phenotype. In GP the mutation operator can modify the topology/structure of the tree or also the content of nodes.

mutation changes a topology of the tree

Let us focus on content at first. The tree consists of inner and leaf nodes. In inner node, there is some operation O_i represented by non-terminal. The mutation operator can switch this non-terminal O_i for another non-terminal O_j with respect to its arity ($\text{arity}(O_i) = \text{arity}(O_j)$). In the case when $\text{arity}(O_i) < \text{arity}(O_j)$ the new subtrees must be generated, in the last case $\text{arity}(O_i) > \text{arity}(O_j)$ the redundant subtrees are removed.

mutation changes a content of the node

The content of a leaf node depends on tree specification (see Sec. 6.2). It can be generalized in such way that a leaf node contains terminal t represented by either constant or variable. The mutation of the node with a variable is made by changing this variable for another one from a set of available variables.

Constant leaf nodes have their value v slightly modified by application of mutation operator. There are plenty of possible approaches how to change that value. The new value can be generated from scratch at random. This practice seems not to be ideal because it brings too big changes in genotype. A better way is to modify current value v. In the case of numerical constants, it could be done by adding some number to v or multiplying v by some number. In the situation when a leaf node does not contain numerical value there a method must be defined which modifies the non-numerical value of that node – e. g. replace a character in a string by another one. The *preserving tendency of mutation*

good practice is to use a *momentum* for modification of numerical leaf nodes. A momentum δ_i is stored for all appropriate leaf nodes. Using of momentum preserves tendency of previous changes of given value. This momentum δ_i is initialized as follows:

$$\delta_i = (-1)^r \cdot \frac{1}{3} \nu_i$$

where r is chosen at random from set $0, 1$ and ν_i is the initial value of node i. After each application of mutation operator to a leaf node, a node value is updated by the addition of the current value of δ_i. Immediately after that δ_i is updated by 20 % of its value randomly.

6.1.3 *Crossover operator*

crossover on trees faces the problem of bloating

The crossover in genetics means the exchange of parts of genotype. This principle is a recombination of information where two or more individuals share their genotype to produce a divergent posterity. In GP the crossover operator selects a node in a tree from the first individual and switches the subtree which has the selected node as a root node with another one from the second tree. The main problem arises from the fact that small change in a tree (especially in levels closer to the root node) causes considerable changes in phenotype. The second challenging issue is known as *bloating* or code growth. It was well studied in recent years [13, 54–56, 77, 78]. It can easily happen that switches of subtrees with distinct depths (for the definition of tree depth see Sec. 6.2) leads to enormously deep trees – this is exactly what *bloating* is. Bloating isn't obviously limited to depth, but the trees can also suffer from bloating to the width. We tried to prevent bloating of the tree by selecting the most compatible subtrees considering their depth and used attributes in leaves. Section 6.3 is dedicated to the problematic of subtree similarity.

6.1.4 *Adaptive evolutionary operators*

The parameters of GP or its components can be changed during the evolution. The methods of changing this parameters can be classified into three main classes – in the first one deterministic parameter control takes place when parameters are modified by some deterministic rule. The next class can be named adaptive parameter control; the principle involves using some feedback from the search as an input for a method of parameter modification. The last group is called self-adaptive parameter control, and it could be said that it is an evolution of evolution – the space of parameters is searched by *meta-evolution*. For a closer overview see [27].

benefit well-behaving operators, punish those with poor results

The self-adaptation is deeply inspected in [81]. In this work, we incorporated self-adaptation approach to mutation and crossover operator. In our implementation of GP, there is defined probability P_o

for each operator which determines how probably is the application of a given operator o to the individual. These probabilities are the subject of estimation (before evolution starts) and adaptive changes during the evolution run.

The general advice is to set P_o of mutation operator up to values around ≈ 0.1. There is a little bit difficult situation with the crossover operator. Probability depends strongly on implementation, encoding, the fact whether the crossover is limited by the number of newly created descendants or not and finally on designer intention if they grant the right to individuals to mate and hand their genotype over. The idea behind dynamic changes of p_o is that well-performing operator will be applied with higher probability. The use of self-adaptive operator probabilities is presented in the section 7.3.

6.2 tree structures

formalization of trees

A tree is a graph with specific requirements. Lets start with a formal definition of a tree as usual in the graph theory [23]:

Definition 4 *A graph* $G = (V, E)$.

Where the V is a set of vertices; therefore, the elements of $E \subseteq [V]^2$ are 2-element subsets of V. The elements of V are the vertices (also nodes) of graph G, the elements of E are known as edges.

Definition 5 *The tree* T *is an* acyclic *graph which is* connected*.*

That means such T contains no cycles and has only one component.

Definition 6 *An acyclic graph has no cycles.*

For a rigorous formal definition of cycles in graphs see [23]. For our purposes, it can be stated that a cycle is a path in the graph which has length $l \geqslant 3$ and when you start to follow that path step by step from its arbitrary vertex x you will come back to x after l steps.

Definition 7 *Connected graph is a graph* G *in which any two of its vertices are connected by a path in* G*.*

For convenience, we consider only directed trees. In a directed tree each edge has its initial and terminal vertex. Each node v has associated property of *indegree* $d^-(v)$ which means how many edges has a terminal vertex in the given node. Analogically there is a property of v called *outdegree* $d^+(v)$ which is equal to the number of outgoing edges from v.

Definition 8 *Leaves in a tree are the nodes* v *with* $d^+(v) = 0$*, each other node is the* inner node*.*

The *root* is one of the nodes of a tree with $d^-(v) = 0$. There are examples of a small tree with named node types on figure 10.

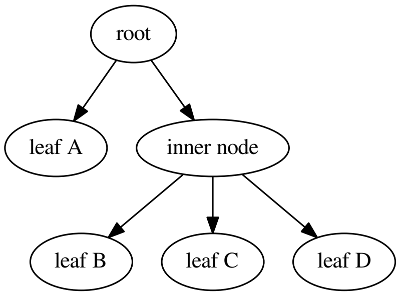


Figure 10: Illustration of all possible types of nodes in a tree

6.2.1 *Decision trees*

Decision trees are the utilization of tree structures for making a decision. The idea is that inner nodes are conditions where interpreter has to evaluate some relation and continue to one of the branches. On figure 11, there is depicted an example of the decision tree.

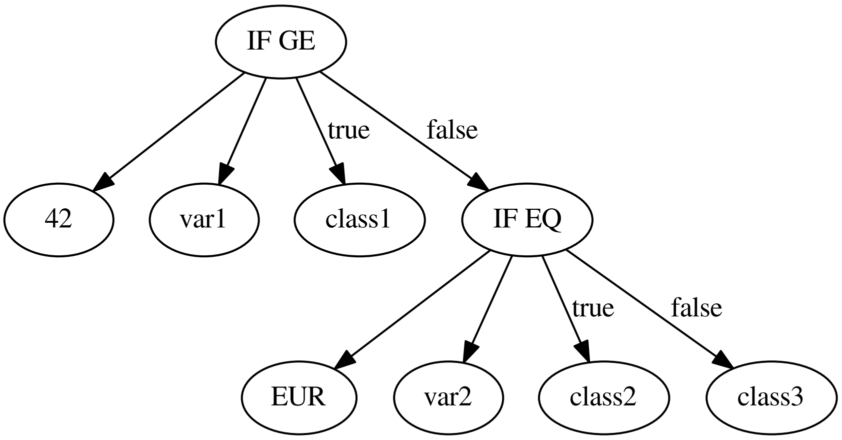


Figure 11: Illustration of a decision tree

In this work, we assume only following kind of inner nodes of a decision tree. The nodes which are the first and second in order from left are always leaves. There is a constant in the first child node; the second contains a variable which is substituted at the time of evaluation. The third child represents true branch and fourth false branch respectively and could be one of the following two types: the final class or another inner node with a relation. The described decision trees are evaluated from the *root*.

6.3 TREE SIMILARITY METHODS

crossover often breaks the phenotype of the trees

There are many problems with application of a crossover operator to the tree structures describing genotype. In most cases, we simply destroy the phenotype (meaning of genotype). This is the reason why a standard simple crossover (swaps randomly selected subtrees) is often not the preferred operator in GP. The impact of operator application is too strong in many cases.

Due to this fact many alternative crossover operators were presented – context preserving crossover by D'haeseleer [21], a modification which measures the performance of subtrees [42] or the one which tries to measure structural similarity [9]. The common element of all mentioned work is to disallow a breaking of a good building block.

The idea behind this research was following: *Take no care of building blocks but try to introduce only slight changes in phenotype.*

tree similarity measures

We present six methods how to measure compatibility of two subtrees. These methods are compared with random approach which serves as a baseline. The random approach switches two randomly selected subtrees each from different parent individual.

The methods will be shown on following examples of trees as depicted in figure 12.

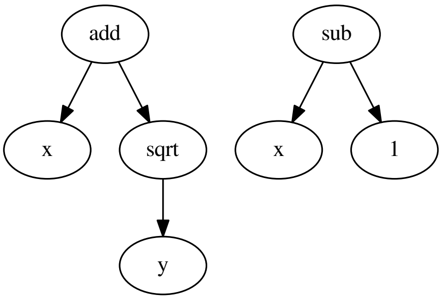


Figure 12: For a description of tree similarity methods we use following trees: $T_a = \text{add}(x, \text{sqrt}(y))$ and $T_b = \text{sub}(x, 1)$

The *Average Arity (AVA)* method computes the average arity of functions in inner nodes. An arity α is the number of operands of the function – in the tree representation, it is an amount of children nodes. The tree T_a has inner nodes with *add* and *sqrt* functions with arities $\alpha(\text{add}) = 2$ and $\alpha(\text{sqrt}) = 1$ respectively. $\text{AVA}(T_a) = (2+1)/2 = 1.5$, $\text{AVA}(T_b) = 2$.

The *Common Variable Set (CVS)* method tries to count how many variables in leafs of both subtrees are the same. In other words, it counts the size of the set originating from the intersection of the sets of leaf variables in both trees. $\text{CVS}(T_a) = \text{CVS}(T_b)) = 1$ because the only one variable (x) is common for both trees.

The *Common Operator Set (COS)* method does the same as CVS method, but instead of leaves' variables, it takes into account the functions (operators) from inner nodes. In our example: $\{\text{add}, \text{sqrt}\} \cap \{\text{sub}\} = \emptyset$ and therefore $COS(T_a) = COS(T_b) = 0$.

The *REV* method tries to represent the value of the subtree. It pushes 0 to all variables in the tree, evaluates the tree and retrieves a value from the root node. This value is now substituted to all variables and so on for 20 times. Final root value is taken as a representative of the given subtree. The algorithm of the REV method is outlined in Listing 5. $\text{REV}(T_a) = 0 + \sqrt{0} = 0, \text{REV}(T_b) = -20$.

Listing 5: A pseudoceode of REV method for subtree matching

```
rev(tree, val, max_depth):
  var_leaves ← tree.terminals \ tree.terminals_with_constants
  foreach l in var_leaves:
    l.value ← val
  tree_value = eval(tree)
  if max_depth >= 20:
    return tree_value
  else:
    return rev(tree, tree_value, max_depth+1)
```

The *Vector Evaluation (VEV)* method takes a randomly generated set of 20 numerical vectors $v_0, \ldots, v_{20}$. The length of these vectors corresponds to the number of unique variables in compared trees – in our case of T_a and T_b it is 2 (for x and y). The tree is evaluated for each vector v_i. Thus we obtain a vector r of 20 results for each of the given trees T_a, T_b. Next, the VEV method computes the distance δ of these vectors by:

$$\delta(r^{T_a}, r^{T_b}) = \sum_{i=0}^{i<20} (r_i^{T_a} - r_i^{T_b})^2$$

This method is closely similar to the method Sampling Semantics Distance (SSD) proposed in [85].

The *String Similarity (STR)* method is based on a string serialization of a tree. The given trees T_a, T_b are represented after serialization as S1 = "add$(x, \text{sqrt}(y))$" and S2 = "sub$(x, 1)$". Next, the Levenshtein distance [100] of these two strings is computed – $\text{STR}(T_a, T_b) = 10$.

The comparison of the performance of presented tree similarity methods can be found in the section 7.4.

7 EXPERIMENTS

The content of this chapter is based on the original publications of the author of this thesis submitted and accepted to international conferences or peer-reviewed journals during the years 2010–2017. With regard to proposed outline of our approach in Sec. 1.2 and also in accordance with the dates of publication, the sections are ordered as follows: A Single-criterion Optimization, Multi-criteria Optimization, Subtree Similarity in Genetic Programming and Auction Systems in Task Distribution.

The advantages of selected form of joined papers are following: The topics are presented to reader as particular steps guiding him through the thesis. Beginning from a verification of basic ideas, following by incremental contributions to the solved problem and finishing by the join of partial sub-problems to the whole solution and comparison of proposed approaches. Also the natural flow of the time during the work is respected.

Some inconvenience for reader is caused by repeating similar topics in introduction parts of each paper. But we think that pros overcomes the cons of selected form.

7.1 SINGLE-CRITERION OPTIMIZATION ON DATAMINING TASKS

The following research was focused on single criterion optimization in parallel solving of data-mining tasks. The objectives were to solve all tasks of given dataset concerning the shortest time, smallest error or a weighted combination of these measures.

The goal of this partial subproblem is to verify evolved trees as a control mechanism in the experimental setup with small artificial tasks being solved concerning one simple criterion. The number of task types and models will be limited to a minimal amount. The combination of the average processing time of each task and the total time of the experiment run will be used as the criterion.

This article was published at International Conference on Soft Computing MENDEL in 2011.

GENETICS IN DECISION BEHAVIOR OF COMPUTATIONAL

AGENTS

Abstract: *We developed system based on JADE framework. This experimental multiagent system consists of two types of agents. Manager distributes tasks to workers, who can decide whether to accept or reject the task. The decision is made as evaluation of expression tree which takes into account agent-task compatibility and attributes of currently solved tasks. The trees are evolved by genetic algorithm and fitness function reflects time consumed for solving the whole task set and average time per one task. The shortest times are achieved for short agent's queues. Best evolved expression tree leads to average time per task 1013 ms. If decided randomly, the most compatible task is solved in 4150 ms.*

Keywords: *genetics, agent, computation, multi-agent, decision making, decision trees*

7.1.1 *Introduction*

An agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives [93]. Main features of an agent are interaction with other agents and adaptivity to enviroment states.

Very often a complex problem could be divided into a set of subtasks. We cannot in principle control central assignment of individual subtasks to agents in dynamic computations systems or environment. For example the manager agent cannot know every local variables and conditions of all worker agents, so he cannot prepare optimal plan for task delivery. There is only a chance to optimize general criteria of complex problem by controling computation effectively in bottom-up direction.

7.1.2 *System architecture*

We developed independent module to experimental Multi-Agent computational System (MAS) [91]. This system named Pikater is based on open source JADE [43] framework written in Java. The system consists of several types of agents. The first one is "manager" agent which offers computational tasks to another type of agents – "workers". Manager communicates with workers, who are answering him, but they don't communicate among themselves. Every worker agent solves tasks in his incoming buffer.

Our module now brings "brain" to worker agents. A worker have now two options when a new task is received. In past he had to accept all incoming tasks. Now, he can either accept a task and store it into buffer or reject it. In both situations he sends a message with his decision back to manager. With this artificial brain the worker agents can make decision, based on local conditions and variables, whether to accept offered task from manager or reject it.

7.1.2.1 *Computing tasks*

Computation task is now an artificial dummy task with a given class of specialization. Each worker agent has its own class of specialization. The distance between agent's and task's specialization (compatibility) affects number of computational steps necessary to complete the task.

It is most efficient when agent has the same class of specialization as solved task. For example if agent is member of class A and task being solved is from class D, the final compatibility c can by shown as:

$$\begin{aligned} c_{A,D} &= \mathrm{diff}(A, D) \\ c_{A,D} &= 1 + |\mathrm{intval}(A) - \mathrm{intval}(D)| \end{aligned}$$

Where intval is function which converts a class type to an integer representation as shown in table 2.

Table 2: The conversion of class type to integer is used in diff function which measures agent–task compatibility. This table shows integer value for each class.

Class of specialization	A	B	C	D	E
Integer representation	1	2	3	4	5

Each task computation can be interrupted when a new task request is accepted. In this case, the new task is enqueued by receiving worker and currently solved task is thrown away. It doesn't guarantee that the accepted task will be computed as the first. Buffer is FIFO, tasks are taken from the head of queue. Next the manager is informed that given task should be offered once more to any worker. All workers have the same probability to be selected as task receiver.

The real-world tasks, such as some data mining or classifying problem have no explicitly defined class. It is hard to tell which method encapsulated in computing agent (eg. Neural Network, Decision Tree) is the best for current task. It will lead to requirement of exploration of task's and agent's metadata for computing agent-task compatibility.

7.1.3 *Agent control*

The worker agent has three basic parallel behaviours. The first one handles communication with manager, next one runs when agent has no active computation and takes the first task from agent's buffer (if there is any) and starts new computation. The last one is computing loop core which makes one step from current task.

7.1.3.1 *Communication and decision making*

Communication part of behaviour is invoked when a message is delivered to worker agent. The agent has to make decision whether a task will be accepted. Based on this, agent sends proper answer message to the manager.

The decision is made as evaluation of an expression which connects weighted attributes of worker agent. The expression is represented by a tree. The main attributes in the expression are number of tasks solved by agent, computed percentage of actual task, agent-task compatibility of actual task, expected time to finish actual task, a load (average expected solving time per task in agent's task buffer) and "happy", which means an average compatibility of enqueued tasks in agent's buffer. Except the first one (long) are all variables real numbers. Shown variables are substituted into the tree at every run of decision process (when new task arrives) and the tree is evaluated as one real value. The result of decision is a boolean variable – accept or reject the offered task. The task is accepted when expression result is greater or equal to zero.

7.1.3.2 *Computing loop*

The computing loop is second main part of worker's behavior. In each iteration some part of task is solved and appropriate agent's attributes are actualized. As stated before, the loop can be interrupted and task can be thrown out. It is desirable to discover possible agent-task incompatibility as soon as possible. Completion of an extremely incompatible task costs more then stopping it and disposing it to a more compatible worker. But on the other hand, the interruption of current task is not good deal when worker agent has no other task in buffer.

This behaviour is planned by internal mechanism of JADE framework. Next run is scheduled by simple command at the end of each iteration of solving loop.

7.1.4 *Genetic evolving of expression trees*

The trees which represent decision expression are evolved by common genetic algorithm [51] for 300 generations. We used tournament selection. Size of population has only about 30–50 individuals because calculation of the fitness function is extremely expensive. It is necessary to reset all agents in MAS enviroment, prepare new task set and computation must race through it.

The number of tasks is constant for whole experiment and it was set up to 100 tasks. It is extremely small amount and there is considerable impact of random element. That means, if we have "luck", then most of tasks are send to suitable worker agents and they solve all tasks in short time. In this case the value of fitness function is significantly higher then in the other case, when major part of task is solved by less compatible agent. But bigger task set would lead to enormous experiment time (several days).

7.1.4.1 *Construction and recombination of expression trees*

The population of decision trees is initialized randomly. Each individual tree has all mentioned agent's variables in its leaves. Each inner node represents one of the functions: ADD (addition), SUB (subtraction) or MUL (multiplication).

Cross and mutation operators are applied directly to the object of tree structure – there is no encoding such as binary for example. Every leaf node is a real number. Therefore it is important to apply operators very carefully, especially crossover operator.

Classic crossover between two individuals is not suitable, because it may produce invalid ancestors. For example there could be two same variables in the new expression tree. And therefore a crossover operator is applied to only one tree and exchanges two subtrees. In consequence it is a kind of mutation.

The application of mutation operator is different for leaf and inner nodes and also has different probabilities in each case. The mutation of inner nodes changes randomly function (ADD, SUB, MUL) in given node and is significantly less frequent than mutation of leaf nodes. The leaf node mutation adjusts value in node by small random number from interval ±10 % of node value.

7.1.4.2 *Fitness function*

The fitness function combines two main criteria of experiment. The first is an average task time T_{avg} and consists of time spent in computing loop and time for which the task waited in buffer. The second one is total experiment time T_{exp} which includes computation and waiting time for all tasks. The fitness function can be computed as:

$$f = \alpha \cdot \frac{R_{exp}}{T_{exp}} + \beta \cdot \frac{R_{avg}}{T_{avg}}$$

where α, β are coefficients with actual values $\alpha = 2$, $\beta = 1$. We put higher weight on total time. Next R_{exp} is empiric average value of total experiment obtained by 100 times run experiment with disabled "brain" – the task acceptation was random with probability 50%. In a similar way R_{avg} is average time per one task given from the same experiment.

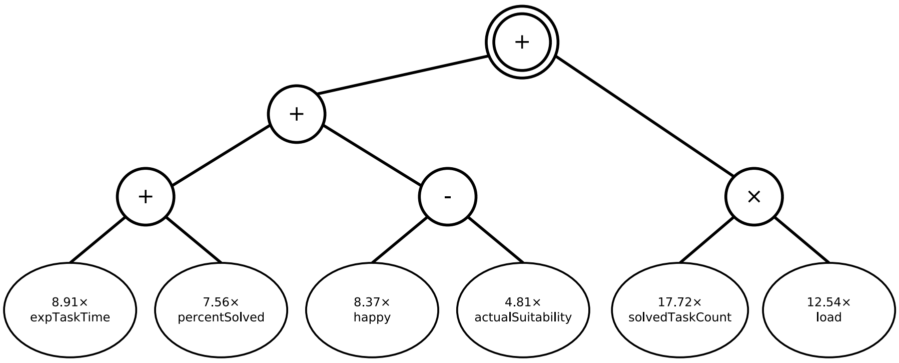


Figure 13: The best evolved expression tree shown on this figure connects certain agent's variables in a polynome. By substituting real values into the tree at decision process we obtain answer whether worker agent should accept offered task or not.

Both R_{exp} and T_{exp} depend on task count. But fitness evaluation counts with it. It follows that random solution with accept rate 50% has fitness very close to 3.

7.1.5 *Experiments and Results*

Experimental task set consists of 100 tasks with random uniformly distributed type class. There are 5 classes and each of 5 workers is member of exactly one class. The most compatible task is solved by an agent in approximately 4150 ms. Regardless to agent–task compatibility the average time per task is 12450 ms. In both cases waiting time (in agent's queue) of task is included. Without waiting time it is about 950 ms for the most compatible task, 2850 ms for average compatible task.

For comparison of decisions based on evolved expressions we run the same task set on MAS, where worker agents make decisions randomly. They accept offered tasks with some level of probability.

Main objective is average elapsed time per one task (waiting in queue included) and total consumed time to solve the whole task set. As we can see from table 3, the total time depends approximately linearly on average task time.

Best evolved expression tree (shown at figure 13) has fitness 36,47158 and leads to average time per task of approximately 1013 ms. That means a very short waiting time of tasks. It is about 12 times better than random 50% accept solution. This perfect time is also caused by good agent–task compatibility during experiment. Simply each task was sent to suitable agent to solve. With many times repeated measurement the time grows to values around 3000 ms.

Table 3: Average computation times

decision method	∅time per task [ms]	time for task set [ms]
accept 2% of tasks	1 004	99 432
accept 10% of tasks	8 593	882 961
accept 50% of tasks	12 450	1 458 572
accept 90% of tasks	16 299	1 648 996
accept 100% of tasks	16 453	1 691 372
best expression	1 013	120 636

7.1.6 *Conclusion*

The shortest times are achieved when worker agent's queues are very short. It leads to minimization of waiting part of task solving time. The best evaluated expression tree after 300 generations has approximately the same quality as situation when 98% of the tasks are rejected by worker agent in order to achieve the shortest possible queue.

In the future, we want to try to modify worker agents to solve some real tasks. There are many options which variables are suitable to be included in decision expression. Some variables could lead to better decision making.

7.2 single-criterion optimization on real datamining tasks

The following natural step was to replace artificial tasks from the first experiment with real ones which are more complex. These tasks have larger datasets and will be solved by real data-mining models (classification, regression). If necessary, the set of worker agent's attributes and also task's attributes will be revised. We stay at single criterion optimization. As the criterion, the mean error of processed task will be used. Next incremental step is the investigation of the impact of the crossover operator. We try to answer the question whether it is possible to turn off the crossover or will it be better to let it active and focus on its implementation.

This article was published at Eighth International Conference on Intelligent Computing (ICIC) in 2012.

Evolving Decision Strategies for Computational Intelligence Agents

Abstract: *An adaptive control system for computational intelligence agent within a data mining multi-agent system is presented. As opposed to other approaches concerning a fixed control mechanism, the presented approach is based on evolutionary trained decission trees. This leads to control approach created adaptively based on data tasks the agent encounters during its adaptive phase. A pilot implementation within a JADE-based data mining system illustrates the suitability of such approach.*

7.2.1 *Introduction*

An agent is a computer system situated in some environment that is capable of autonomous action in this environment in order to meet its design objectives [93]. *Autonomy* is used to express that agents are able to act (to perform actions) without the intervention of humans or other system. Agents' important features are adaptivity to changes in the environment and collaboration with other agents. Interacting agents join in more complex societies, *multi-agent systems* (MAS). These groups of agents gain several advantages, as are the applications in distributed systems, delegacy of subproblems on other agents, and flexibility of the software system engineering.

An *intelligent agent* is one that is capable of *flexible* autonomous action in order to meet its design objectives, where flexibility means three things: *pro-activeness* (goal-directed behavior), *reactivity* (response to changes), and *social ability* (interaction with other agents).

7.2.2 *Computational MAS*

In our approach, a computational MAS contains one or more computational agents, i.e. a highly encapsulated objects embodying a particular computational intelligence method, and collaborating with other autonomous agents to fulfill its goals. Several models of development of hybrid intelligent systems by means of MAS have been proposed, e.g. [104] and [59].

A *computational agent* is a highly encapsulated object realizing a particular computational method [60], such as a neural network, a genetic algorithm, or a fuzzy logic controller. The main objective of our architecture is to allow a simple design of adaptive autonomous agents within an environment of a computational multi-agent system. In order to act autonomously, an agent should be able to cope with three different kind of problems: cooperation of agents, a computation processing support, and an optimization of the partner choice. The architecture we present is general in the sense that it can be easily extended to cope with different problems than those mentioned, nevertheless, we present its capabilities in these three areas.

Cooperation of agents: An intelligent agent should be able to answer the questions about its willingness to participate with particular agent or on a particular task. The following subproblems follow: (1) deciding whether two agents are able to cooperate, (2) evaluating the agents (according to reliability, speed, availability, etc.), (3) reasoning about its own state of affairs (state of an agent, load, etc.), (4) reasoning about tasks (identification of a task, distinguishing task types, etc.).

Computations processing: The agent should be able to recognize what it can solve and whether it is good at it, to decide whether it should persist in the started task, and whether it should wait for the result of task assigned to another agent. This implies the following new subproblems: (1) learning (remembering) tasks the agent has computed in the past (we use the principles of case-based learning and reasoning — see [4], [2] — to remember task cases), (2) monitoring and evaluation of task parameters (duration, progress, count, etc.), (3) evaluating tasks according to different criteria (duration, error, etc.).

Optimization of the partner choice: An intelligent agent should be able to distinguish good partners from unsuitable ones.

So, the architecture must support *reasoning*, *descriptions* of agents and tasks (we use ontologies in descriptions logics — see, e.g., [6]), *monitoring* and *evaluation* of various parameters, and *learning*.

Our architecture deals with this problem by combining a fixed set of basic monitors of agent and system state, and fixed set of possible actions, with a model created by means of decission trees evolved by genetic programming based on the performance of the system. Data, meta-data, and data-mining algorithms are described in a domain ontology KDDONTO [22], based on description logics. This ontology is also used in other part of the system for automatic composition of algorithms forming valid data-mining processes.

7.2.3 *Control of Computational Agent*

Two types of agents in our computational MAS are important when considering the agent adaptive control task the *computational agent*, whose role, as a worker, is to accept or reject offered tasks from the *manager*. Each accepted task should be solved with regard to minimize its error and at the same time, to optimize the overall time complexity. The manager distributes tasks to worker agents in a non-informed manner, i.e. it does not target specific task to specific agent, rather a random worker is selected from the pool of potentially available agents. A computational agent can accept more than one task. In this case it stores the accepted task in a FIFO buffer. The size of this buffer is selected with regard to the total number of tasks in one experiment run. In an extreme case one worker could solve all tasks in one run, but this is hardly optimal.

The computational agent has to make a decision of task acceptance each time when the manager offers a new task. Good decision making leads to acceptance of tasks that are suitable for given agent, and rejection of others. With growing impact of time criteria in quality of the solution, the worker should possibly reject even suitable tasks in order to shorten number of its tasks in buffer. The problem of task acceptance is thus dependent on the inner state of computational agent as well. Such state can be described by set of attributes with integer and real domain.

We selected [74] these *attributes* as the important indicators of agent state: number of tasks solved by an agent, computed percentage of the current task, agent-task compatibility, expected time to finish an actual task, a "load" (average expected solving time per task in agent's task buffer), and "happiness", which means an average compatibility of enqueued tasks in agent's buffer.

These attributes are the basis of the decission formula evolved by means of genetic programming algorithm. Thus, we are seeking a general polynominal, where each attribute can be weighted by a real coefficient, and these blocks ($w_i * \mathtt{Attrib}_i$) are combined together in a general manner by addition, subtraction and multiplication operations. are subtraction, addition and multiply (*SUB*, *ADD*, *MUL*).

This polynomial is internaly represented by a binary tree where agent's attributes are stored in the leaf nodes (terminals), while inner nodes (non-terminals) represent operations. The operation in the level just above a leaf node is always multiplication in order to obtain the weighted tuples $w_i * \mathtt{Attrib}_i$.

By evaluation of this tree for a particular task, an agent obtains one real value. When this value is higher than zero, the offered task is accepted, otherwise it is rejected. Decision making of acceptance is only one part of agent control which is adaptive, and it depends on actual state and context. All other worker and manager actions can be considered as fixed for our experiment.

7.2.4 *Genetic Programming*

The decision polynomial is represented by a tree as we described in section 7.2.3 and trained by a common genetic programming algorithm as described in [51]. Typically, a popupalation of 30 individuals was evolved for 200 generations. More details about the algorithm follows.

For constructing individuals of initial population, it is important to compute the depth of each randomly generated binary tree, in order to eliminate nodes with only one ancestor. The condition is that every inner node has two ancestors, and every leaf node has no ancestors. The needed depth d can by calculated as:

$$d = \lceil \log_2(\text{count of attributes}) \rceil + 1.$$

It is not neccessary for the deepest level of the tree to be completly filled. As we have mentioned, the level immediately above the leaf nodes contains only multiplications as operators, thus these nodes are *freezed*. It means they cannot be changed during the evolution. Values and attributes for each operator comes from its child nodes. Weight coefficients are choosen randomly from the $\langle 0, 1 \rangle$ with an uniform distribution. Operators (*SUB*, *ADD*, *MUL*) in the inner nodes are chosen randomly.

Once we have the initial population, the main cycle of evolutionary search is started (see figure 7.14(a). Selection of candidates is done by *tournament* with the size of 5 individuals — the winner of tournament is selected for the application of the mutation operator. The probability of mutation is 80 %. While this number is quite high, note that the mutation impacts only one node of the tree. We select at random one unfrozen node and change it. In the case of leaf node, we change its value by adding δ which is calculated as:

$$\delta \;=\; R * \text{oldNodeValue},$$

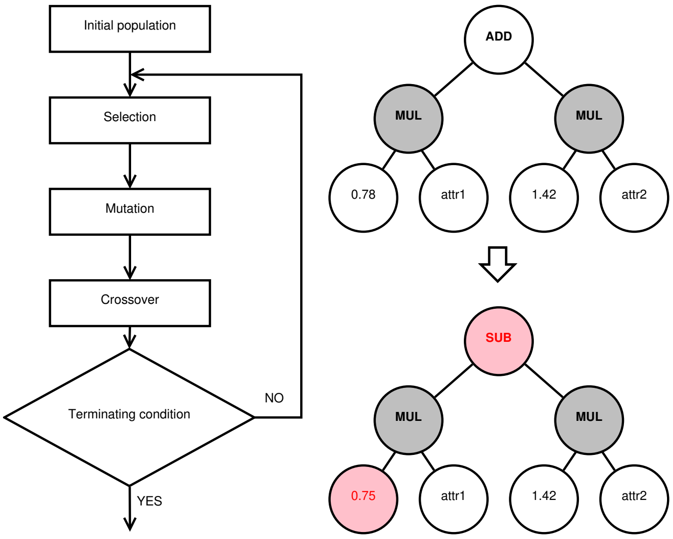
where R is a random number from interval $\langle -0.33, 0.33 \rangle$. For an example of two valid mutations on one tree see figure 7.14(b).

During our experiments we decided to avoid the crossover operator at first since it did not bring any advantage in contrast to the mutation. We speculate that it might be caused by disruption of the complete set of attributes by crossover, and possibly resulted in too incomplete information for inner state. The power of mutation seem enough to sufficient exploration of the state space.

Later we enabled the crossover with 10 % probabality for confirmation of the hypothesis of incomplete information about inner state. It led to evolving trees with greater number of leaves and levels, therefore some attributes were used twice or more and some were omitted.

For faster convergence we implemented the *islands model*[95] which increased the speed of exploration by localized search with asynchronous combination of the best individuals among islands. Elitism was used to attain the overall best individual and preserve convergence. The fitness function reflects model's mean error computed gradually for each combination of task and agent. This mean error is summed over all tasks solved by given agent during one run of experiment. For the case of maximization, we count fitness value as:

$$f \;=\; \frac{1}{E_{mean}}.$$



(a) The main loop of genetic algoritm.

(b) Example of mutation

Figure 14: Genetic algorithm and decision trees.

7.2.5 *Experiments*

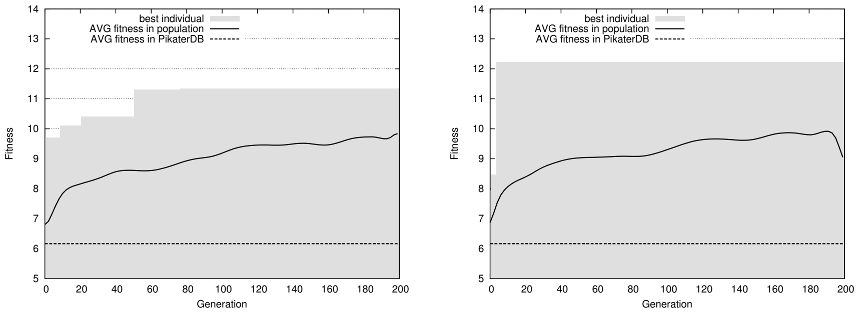
For the experiments we have prepared an environement with 3 worker agents (*Multilayer perceptron*, *Naive Bayes* and *Radial-Basis Function*) and one manager agent described in section 7.2.3. Their aim was to solve 60 tasks which were randomly picked instances of datasets from a subset of the UCI Machine learning repository[1] (*car*, *breast-cancer*, *iris*, *lung-cancer*, *tic-tac-toe* and *weather*). All of them are relatively small to middle-size datasets, where classic models usually achieve good results. This task set is genereted once for each fitness counting.

Due to speed optimalization as described above, we used the database of results precalculated in our Pikater [46] multiagent system project. Since there are potentially many results for the given combination of computational agent and data (varies from 250 to 3990), depending on the setting of agent's options, one result is selected at random.

The configuration of the genetic programming algorithm is as follows: Number of generations is 200, population size is 15, tournament size is 3, elite count is 1, mutation probability is 0.8 and cross probability 0.0 (or 0.1 in the case of enabled crossover).

The whole experiment runs typically for several hours. Simulation needed for obtaining one fitness value takes from 2521 to 43024 ms on a fast desktop machine running Linux. As we can see on figure 15, the average fitness value in population grows approximately from 7 to 11 in 200 generations. Gray columns represent the best individual

[1] Available at `http://archive.ics.uci.edu/ml/datasets.html`



(a) Without crossover

(b) With enabled crossover

Figure 15: Evolution of fitness value.

(elitism), the solid line is an average fitness in the whole population. The constant dashed line represents value of fitness averaged over complete precomputed database of results.

The enabled crossover leads to faster exploration – see fitness value of the best individual on fig. 7.15(b) – and evolved trees can be very distinct and nonuniform. The process of evolution is similar in both cases.

The best evolved tree from all experiments had fitness 20.684, is shown on figure 16 and represents this polynomial:

$$f = (0.375 \cdot expTaskTime - 1.751 \cdot percentSolved) * \\ (1.98 \cdot happy + 0.743 \cdot actSuit) + 0.103 \cdot load * 0.373 \cdot solvedTasksCnt. \tag{1}$$

The meaning of agents attributes is described in section 7.2.3. We can see that some weights of attributes are relatively small as compared to the others. It can be stated that some attributes (such as *load*, and maybe *expTaskTime* and *solvedTasksCnt* as well) have smaller impact on decision making than others (*happy* and *percentSolved*).

In table 4 we can see comparison of decision making methods – all three methods were run three times and the values are averaged. Real accept ratio is a value obtained by dividing the number of cases when offered task is accepted by number of all tasks. The rejected task was returned to the manager and offered next time.

The computational time consists of real time of task solving and time for which the task has to wait in worker's buffer. The pure solving time is the same for all cases in average. Thus, we can state that the best trees optimize second part of computational time – the waiting time. It is done by keeping short queue of waiting tasks.

It is interesting that the best tree evolved with enabled crossover leads to 100 % of acceptance but error is significantly smaller and computational time is not so much higher.

Table 4: Comparison of different decision making approaches

Decision method	Random 50 % accept. ratio	Best tree without crossover	Best tree with crossover
Real acceptance ratio	0.4845	0.4286	1.0000
Avg. task error	0.1733	0.1375	0.0929
Computational time	5248 ms	1749 ms	1895 ms
Avg. value of polynomial	–	26.4870	90.3724

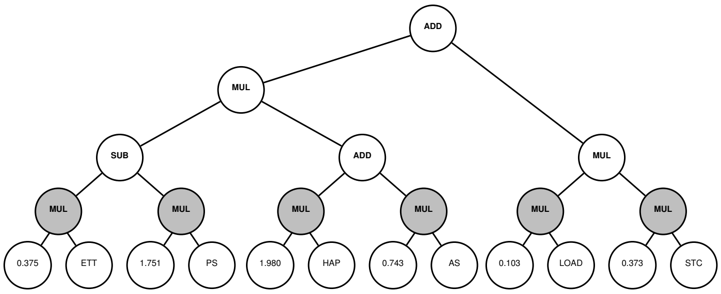


Figure 16: Best polynomial encoded as a tree.

7.2.6 *Conclusion*

In this paper we have described an autonomous agent control method for data-mining multiagent systems. In contrast to our previous work [87] where the agent control has been designed by hand, a genetic programming approach is used to evolve the computational agent inner logics based on the data task sets it encounters. The agents decision to accept or reject given task is based on its state, its load and on the compatibility of the agent with the data set it is supposed to learn. The objective function of the evolution is based on agents performance on the data measured by the traditional least square error.

We have achieved comparable results to the hard-coded agents with respect to the average error achieved on the sequence of tasks. The best evolved individual has reached about three times better perforamce in comparison to randomly deciding agents. At the same time the average duration of the task decreased by tens of per cent. This is an encouraging result as the time complexity is not explicitly included in the objective function of the evolution, it is rather only induced by the possibility to reject incompatible tasks which the agent was able to learn.

Although the results show relevant improvement, there are some problems that remain unanswered so far. In the future work we plan to extend our work in the following ways. The polynomial encoding of the decission tree may be too limited for the real-world tasks, we plan to make use of general decision trees without the frozen nodes described above. It goes hand in hand with tuning parameters of genetic programming such as impact of crossover etc. Another challenge is to include both the time and error rate objectives in the evolutionary algorithm and make use of the multi-objective evolutionary approach to optimize with respect to both criteria. This represents a challenge how to express the duration of the task without considering the speed and load of particular processor, and at the same time how to weight out the ratio between duration and error values. Nevertheless, this represents an important step towards human-competitive design of the machine learning approach to data mining multi-agent system.

7.3 MULTI-CRITERIA OPTIMIZATION ON DATAMINING TASKS

The following step was to use multiple criteria for which the task set will be solved by agents in MAS. We try to combine both criteria (mean error and time) and use single criterion optimization. The next logical step is the utilization of multi-criteria optimization. For that, we use the NSGA-II algorithm [19]. The objectives were to solve all tasks of the given dataset concerning the shortest time and smallest error. These two measures were optimized independently. We try to improve the decision trees with the *if* node.

This article was published at 5th International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA) in 2014.

Multiobjective Genetic Programming of Agent Decision Strategies

Abstract: *This work describes a method to control a behaviour of intelligent data mining agent. We developed an adaptive decision making system that utilizes genetic programming technique to evolve an agent's decision strategy. The parameters of data mining task and current state of an agent are taken into account by tree structures evolved by genetic programming. Efficiency of decision strategies is compared from the perspectives of single and multi-criteria optimization.*

7.3.1 *Introduction*

The main motivation for this work is to explore the topic of adaptive evolution of decision strategies for multi-agent systems. In a real world applications of distributed computational system there is often one central unit whose role is to divide problem and create a plan for optimal distribution to solving units. The next but not the last role of such central unit is to collect solved parts and join it to the final solution of the original problem. Therefore it may be difficult to create this central unit. An intelligent division of a task set is sometimes a difficult problem. And this is the main idea of the proposed article – to transfer a problem of informed distribution of problem parts to solving agents.

An agent is a computer system situated in some environment that is capable of autonomous action in this environment in order to meet its design objectives [93]. This general definitions has to be specified to avoid misunderstanding. The term *autonomy* means that an agent is able to perform its actions without intervention of another system or humans.

Multi-agent systems (MAS) consist of one or more agents and they benefit from adaptivity and collaboration of agents. Such systems are used to solve distributed problems, simulations, or in many domains of computational intelligence. This work applies computational MAS in the data mining field to distributed tasks solving.

An intelligent agent should do more than blindly follow its defined behaiviour. It must be *reactive* (capable of response to changes), *proactive* (executing goal-directed behavior) and *social* which means that agents can interact with other agents in a meaningful way.

7.3.2 *Computational MAS*

A *computational agent* is a highly encapsulated object realizing a particular computational method [60], such as a neural network, a genetic algorithm, or a fuzzy logic controller. Our system is designed to perform distributed data mining task solving. *Worker* agents encapsulate data mining models and their goal is to solve tasks which were randomly picked instances of data sets and blindly distributed by *manager*. Each worker has to make a decision whether to accept or to reject offered task to optimize designed criteria. The architecture we present can be easily modified or extended for application to different problems.

A worker agent should be able to make such decision which leads to acceptance of tasks which promise better results by solving by encapsulated model. It is also important to recognize whether to discard a started task which must be solved by another agent and take a new one instead of the discarded one with perspective to be solved better. This brings new requirements for our agents: an ability to compute tasks suitability for a model, case-based reasoning (see [1]) to remember task cases which agent computed in the past, handling task's and evaluation's parameters (progress of task, elapsed time, system load, etc.).

We utilized a subset of data sets from the UCI Machine learning repository [7]. The selected subset contains these data sets: *car*, *breast–*

cancer, *iris*, *lung–cancer*, *tic-tac-toe*, *weather*). All of them are relatively small to middle-size classification data sets, where classic models usually achieve good results. For the classification itself all available attributes of given dataset are used.

We use next attributes as indicators of agent state: *solvedTasks* – number of tasks solved by an agent, *expSolSteps* – expected steps to finish an offered task, *stepsSolved* – number of solved steps of currently processed task, and *currentSuit* – agent–task compatibility of currently solved task. In our previous work [75] we have utilized a buffer of waiting tasks for an agent and a limited number of attributes. We added these new attributes: *offeredSuit* – agent–task compatibility of offered task, *percentSSol* – computed percentage of the current task, and *ticksToEnd* – expected number of steps to finish actual task.

The evaluation of fitness function for each individual is very expensive in terms of time. Calculation of each fitness needs to run the simulation and solving of a whole set of tasks. Thus we decided to use precomputed results of each considered pair of agent's inner model and task type. This precomputed results we took from our another project – Pikater [46] which is multi-agent data mining system.

7.3.2.1 *Control of Computational Agent*

Two types of agents in our computational MAS are important when considering the agent adaptive control task; the *computational agent* whose role, as a worker is to accept or reject the offered tasks from the *manager*. Each accepted task should be solved with regard to optimization criteria. The manager distributes tasks to worker agents in a non-informed manner, i.e. it does not target specific task to specific agent, rather a random worker is selected from the pool of potentially available agents. A computational agent can accept only one task. If a worker has actually no currently processed task it simply starts to work on the accepted task. On the other side, when it accepts a new task while solving another, it has to discard the task which currently being solved. Such task is returned to the manager to be redistributed and the newly accepted task goes to processing.

The main goal of decision making is to accept such tasks that are suitable for current working agent and to reject tasks which would this agent solve with worse value of optimization criterion – eg. bigger error or greater elapsed time.

When a worker agent receives a new task and has to make a decision, all attributes in the tree are substituted by current values and the tree is evaluated. The value we obtain from root element is compared to the threshold and if it is greater than threshold value an agent accepts offered task. Otherwise the offered task is rejected and returned to the manager for next distribution.

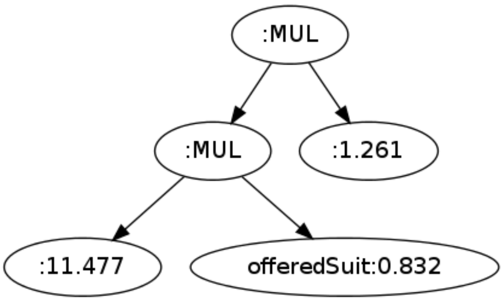


Figure 17: An example of a small tree structure used in working agent for decision making.

7.3.3 *Genetic Programming*

A single criterion optimization is done by common genetic programming algorithm as described in [51]. Further approach with multi-criteria optimization uses NSGA-II algorithm as presented in [19].

An individual represents a tree structure for decision making. The sctructure is a polynomial represented as a tree structure. The leaf elements of that tree contain attributes of computational agent, the solved and also newly offered task. All attributes are represented by numbers from integer to real domain. Inner nodes of the tree represent binary operators such as addition, subtraction and multiplication (ADD, SUB, MUL).

Trees in initial population are generated randomly. The depth of a tree depends on number of attributes. The level immediately above the leaf nodes contains only multiplications to express weigh of each attribute and such node is indifferent to mutation operator. An inner node operator (*ADD*, *SUB*, *MUL*) is chosen randomly and weight coefficients comes from the $\langle 0, 1 \rangle$ with an uniform distribution. Further we show enhanced approach with an *if* operator. Example of a small tree with only one attribute is shown on Fig. 17.

The mutation operator has different behavior in the case of application to the leaf node with real number value and in the case of inner node with a function. Each node with numerical value has also momentum variable δ of same type. The momentum variable is used to preserve tendency of changes. It is initialized according to this formula:

$$\delta = (-1)^r \cdot \frac{1}{3} \nu \tag{2}$$

where r is chosen at random from set $\{0, 1\}$ and ν is initial node value. Thus when mutation of numerical node is applied we modify δ:

$$\delta_{T+1} = \delta_T \cdot \left(1.1 - \frac{\mathrm{rnd}()}{5}\right) \tag{3}$$

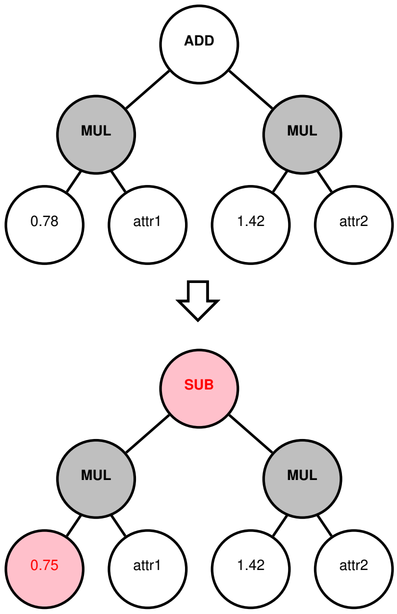


Figure 18: An example of mutation – in inner node and also in leaf node.

where rnd() represents a random real number from interval $\langle 0, 1 \rangle$. This means change by 10 % up or down. The new value δ_{T+1} is simply added to the value of mutated node. Mutation of inner nodes is more simple – we ony select another operator which has the same arity as the old one.

The crossover operator switches subtrees for two randomly selected nodes from two individuals – each of crossed nodes come from a different individual. We tried to prevent bloating of the tree by selecting the most compatible subtrees of the selected nodes. Compatibility of the subtree is measured by ratio of the number of identical attributes in leaf nodes of selected subtrees to the sum of all leaf nodes in these subtrees.

Fitness function expresses quality of each individual. Commonly maps encoded form of individual to a real number. In our both single and multi-criteria optimization experiments we want to maximize the fitness value. Therefore we have used following fitness expressions – error criterion:

$$f_e = \frac{1}{E_{mean}} \tag{4}$$

where E_{mean} is model's mean error and is computed and averaged over the whole task set solved by working agents during one simulation run. Time criterion:

$$f_t = \frac{1}{T} \tag{5}$$

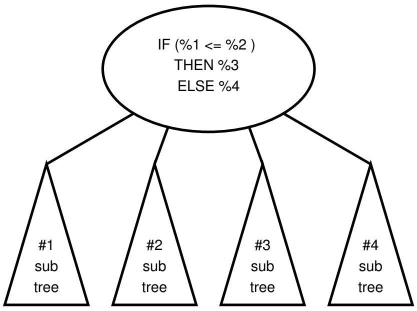


Figure 19: *IF* operator example.

is computed in same way as above, but T is average time spent by a working agent with given task. The last criterion simply combines normalized values of both described criteria:

$$f_{\text{combined}} = \frac{1}{E_{\text{mean}_{\text{norm}}}} + \frac{1}{T_{\text{norm}}} \tag{6}$$

where each particular criterion is normalized. The normalization is based on averaged experimental results obtained for each of the criteria independently. Therefore it is rarely possible to obtain value of f_{combined} higher then 2.0. At first we performed experiments with single criterion (all of eq. 4–6). Naturaly, the next step was multi-criteria optimization and we taken into account criteria eq. 4 and eq. 5 at once.

7.3.3.1 *From polynomial to general expression*

We followed concept of a binary tree structure which represents polynomial connecting attributes and we generalized it. By adding a ternary operator *if* (means condition) we left binary trees behind us and brought more expressive power to our trees – see [30]. See Fig. 19. The *if* construct brings possibility to make several "smaller" decisions based only on some subset of attributes and combine them to the main result whether to accept or to reject the task. The clarity of the meaning of the operators in the inner nodes with arity more than two requires to take care of the order of child elements during application of evolutionary operators.

7.3.3.2 *GP experimental parameters*

We performed many experiments to tune parameters of the genetic programming. After that we performed all the experiments described in chapter 7.3.4 with following configuration. The evolution run trough 300 generations with population of 30 individuals. Selection for crossover was done by tournament selection and size of tournament was 7 individuals. The probability of mutation was set up to 10 % and for crossover was 1 %. Simulation was done 5 times to obtain fitness of one individual. We have applied elitism to prevent the best individual from evolutionary operators. And also *islands model* [95] for faster convergence by localized search with asynchronous combination of the best individuals among islands.

7.3.4 *Experiments*

We focus on impact of each part of proposed approach: the influence of new attributes, how *if* operator changes the evolution and its effect on decision making, and the comparison between single versus multi-criteria optimization.

Our experimental environment contains one *manager* agent and three *worker* agents which encapsulate these data mining models: *Multilayer perceptron*, *Naive Bayes*, and *Radial–Basis Function*. Their goal is to solve 30 tasks which were randomly picked instances of data sets and blindly distributed by *manager*.

During this work we actualized our database of the precomputed results. The actualization means that we imported lots of new precomputed results. At the begining there were 50710 of agent–task results. The new results were filtered to the same subset of pairs agent–task. This enlargement of the database provided us 105121 precomputed results.

7.3.4.1 *Single criterion optimization – Comparison of old and new attributes*

At first let us focus at influence of a new set of attributes. As you can see on both figures 20 and 21, the addition three new attributes (*offeredSuit*, *percentSSol*, *ticksToEnd*) lead to very low effect. The decision making without buffer is harder and rejection of currently solved task is too costly. But newly introduced difficulty is compensated by new attributes.

7.3.4.2 *Single criterion optimization – The impact of* if *operator*

In the following experiments we always use complete set of attributes. On the figure 22 progress of evolution with *if* operator (black lines) and without (gray lines) operators is plotted. It can be stated that with error criterion it is difficult task to evolve any better decision system then randomly generated one – only thanks to elitism better solution was obtained. The average quality of the population is the same as at the beginning. This fact can be demonstrated also in the case of the time criterion (Fig. 23) where it is shown that the average quality of the population grows virtually during all 300 generations. It can be expected that during 1000 generations long evolution the average fitness value over the population will be greater than in the case of the tree structure without *if* operator. On the other hand, with

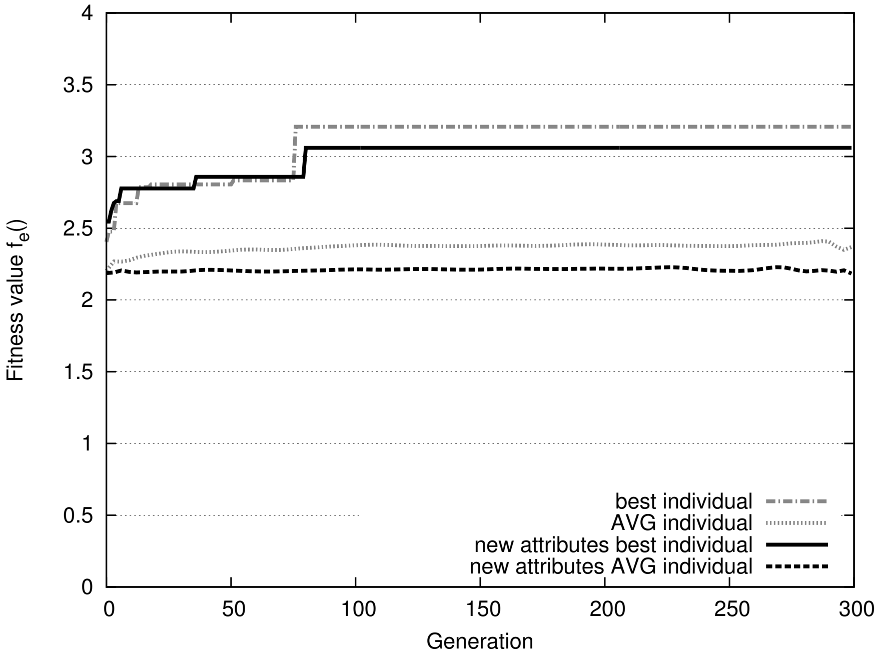


Figure 20: The influence of new attributes – evolution using fitness function f_e (see eq. 4).

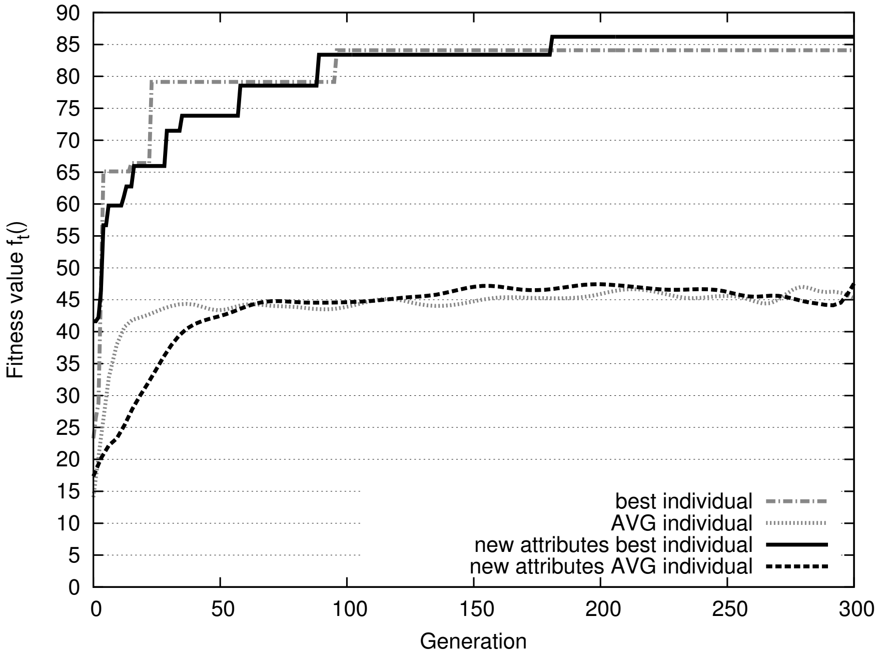


Figure 21: The influence of new attributes – evolution using fitness function f_t (see eq. 5).

complex criterion $f_{combined}$ – see Fig. 24 – the *if* operator has smaller improvement in the first 75 generations and later it produces better average values of fitness function and also better elitist was found.

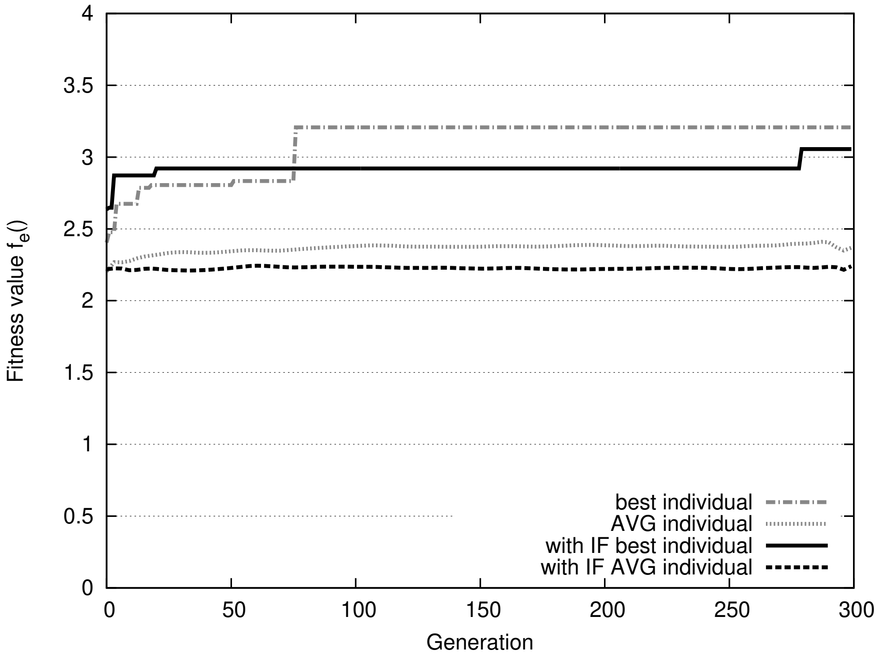


Figure 22: The impact of the *if* operator – evolution using fitness function f_e (see eq. 4).

7.3.4.3 *Multi-criteria optimization (MCO) approach*

The last section of experiments compares single and multi-criteria optimization. The new attributes are present in this experiments; *if* operator is in the set of used inner node functions.

The first results of MCO are plotted on the figures 25 and 26. The vertical and horizontal lines are average and maximal values of each criteria obtained from a single criterion optimization. For error criterion (see eq. 4) the average is 2.26 and the maximum is 3.71. For time criterion (see eq. 5) is the average is 15.46 and the maximum is 42.40. These values are distinct from those depicted on previous figures. This is caused by the fact that our precomputed database was expanded with many new results during our work on MCO experiments.

The whole population is concentrated in the left part of the graph even below the average value of time criterion. For the error criterion we can state that we have achieved the same results as in the single criterion evolution. The reason of failure in time domain comes from the fact that mainly the *Multilayer perceptron* is quite slow and results in the precomputed database have great variance. Thus the extremely distinct values (eg. "1" and "1500") are too big deal for evolution to connect them through attributes derived from the time of task solving.

We solved the problem of very distinct values of time by creating a new meta parameter *logTime*. The value of this parameter is computed

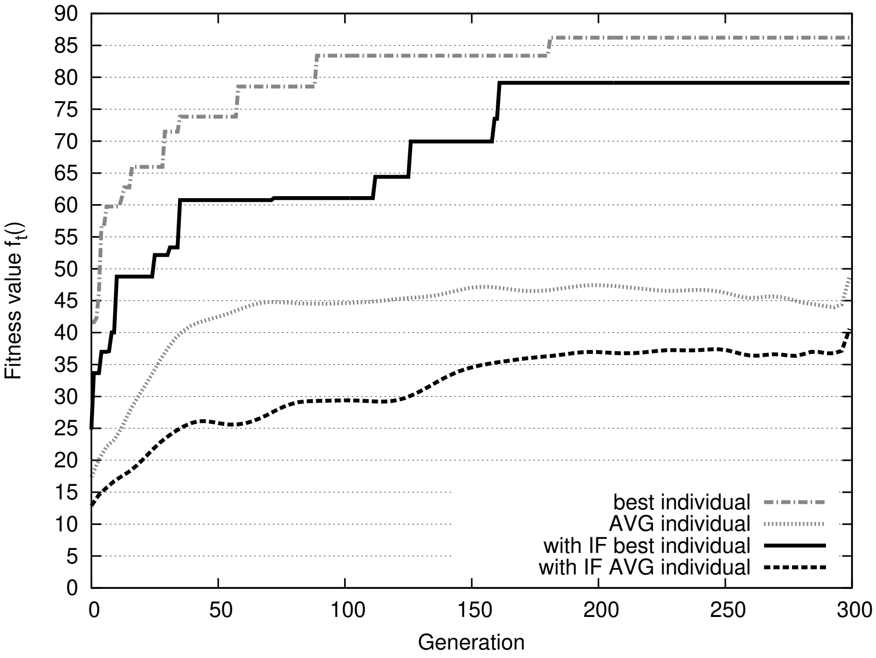


Figure 23: The impact of the *if* operator – evolution using fitness function f_t (see eq. 5).

Table 5: The summary of the best results of each experiment

experiment	mean squared error	time [ticks]
SCO old attributes	0.3118	1.0120
SCO new attributes	0.3267	1.0117
SCO *if* operator	0.3272	1.0127
MCO all from 1^{st} front	0.3175	1.0215
	0.4273	1.0143
	0.4935	1.0095

by application of natural logarithm to the value of elapsed time from precomputed DB and a small shift:

$$p_t = \ln t + 1$$

where p_t is a new meta parameter and t is the original time. This transformation maps values of original parameter from $\langle 1, 1500 \rangle$ to $\langle 1, 8.31 \rangle$ which are better for connecting them by decision making tree structure. The result is shown on figure 26. It can be concluded that population expanded over the average value of time criterion and even that we found several better individuals in the terms of time criterion than in the single criterion optimization. The best evolved tree from multi-criteria optimization with meta parameter *logTime* applied instead of the original task parameter *time* is shown on Fig. 27.

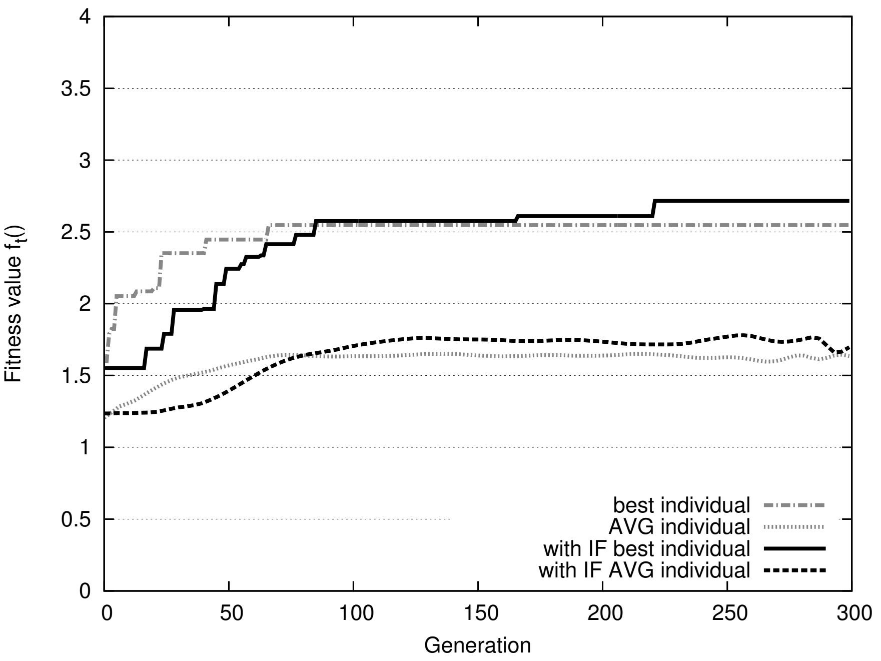


Figure 24: The impact of the *if* operator – evolution using fitness function f_{combined} (see eq. 6).

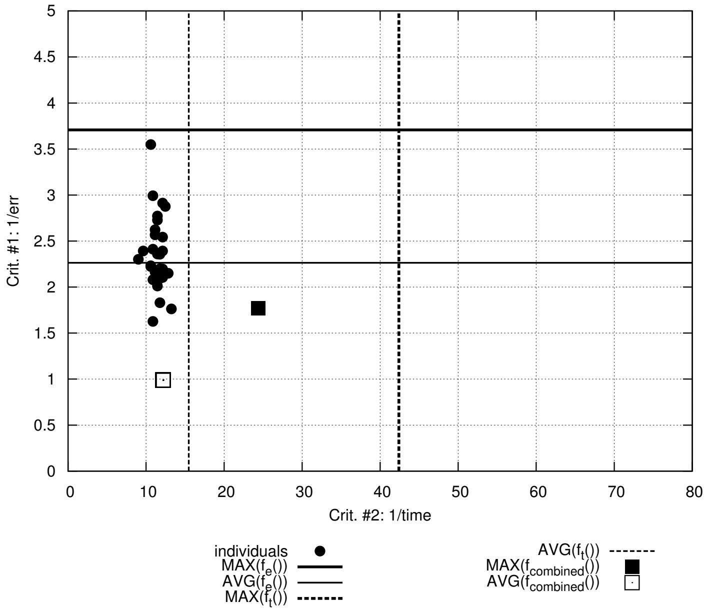


Figure 25: The population after 300 generations of multi-criteria optimization with original time parameter.

The table 5 shows the best evolved individuals from each experiment. The error criterion was transformed to the original value of

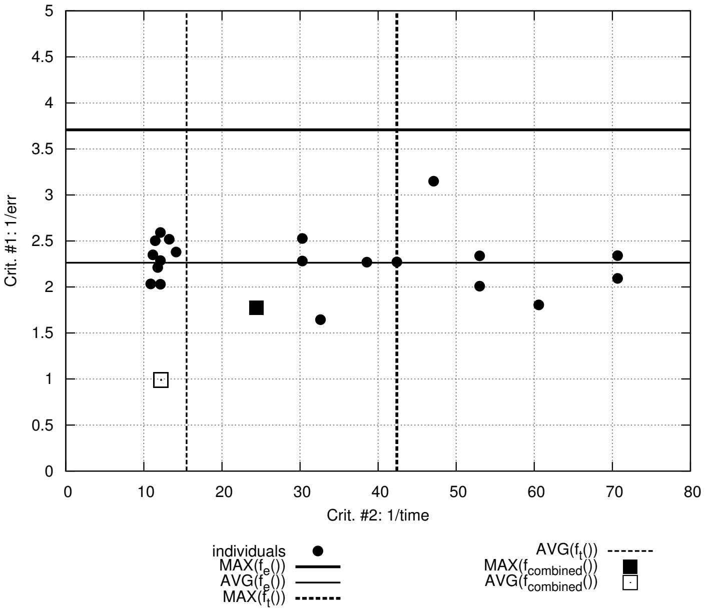


Figure 26: The population after 300 generations of multi-criteria optimization with meta parameter *logTime* applied instead of original time value.

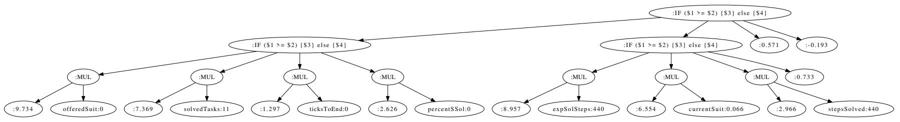


Figure 27: The best evolved tree from multi-criteria optimization with meta parameter *logTime* applied instead of the original time value.

mean squared error. The same process was applied on the time criterion. The repeated measurements of the best individuals and averaging of results caused the fact that ticks are real numbers not integers.

7.3.5 *Conclusion*

This work deals with evolution of the decision making system for autonomous computational data mining agents. The evolved tree structures connect agent's and task's attributes and serve as a control mechanism for worker agents. The optimization criteria concern error and time domain.

The new set of attributes preserves quality of the population in both criteria. Moreover, we added *if* operator to the set of available inner node operations. In the case of single criterion optimization this change caused slightly worse performance for experiments which used fitness function expressions as described by Eq. 4 and Eq. 5.

On the other hand with the more complex fitness function defined by Eq. 6 the results of individuals with *if* operator achieved better results.

The multi-criteria optimization provided comparable results in the error criterion; however, in the time criterion the results were below the average. The problem was caused by very distinct values of the time attribute. The task duration varies with different parameter settings in the orders of magnitude, therefore the logarithmic transformation was natural choice. The results were improved after addition of new meta parameter which is derived from original time by the application of logarithm function. The improvement is manifested on the case of the time criterion where the best individuals overcame the elitist from single criterion optimization.

7.4 SUBTREE SIMILARITY IN GENETIC PROGRAMMING

In this section we defined our goal as to explore the possibilities of improvement of crossover operator in GP. In evolution, we use the self-adaptive mutation operators which change the probability of their application depending on their performance. Furthermore, we focus on search of the best method for matching subtrees for the crossover operator. The verification and performance measures will be done on the problem of symbolic regression on several benchmark datasets.

The following article was published on the 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) in 2017. The work is a result of hundreds of experiments with GP and many tries to found the best parameters and operators for the evolution of the trees. The most interesting challenge was to minimize the negative impact of crossover operator to the selected trees which is a known problem. We proposed several methods to find the matching subtrees for exchange during a crossover operator application.

Matching Subtrees in Genetic Programming Crossover Operator

Abstract: *In this paper we study techniques that should reduce the destructive impact of crossover in genetic programming. The quality of crossover offsprings is often lower than ancestors due to the fact that a small change in individual's genotype tree structure has a great impact to its phenotype. Therefore we propose and test several methods for matching subtrees to find the best possible cutting point for crossover of trees. Our approach utilizes the adaptive probability of operators with the intent to reinforce the well-performing operators. A relation to the semantic genetic programming approach is also investigated. The experimental results show that the average arity based technique performs best from the proposed methods.*

7.4.1 *Introduction*

The primary goal of the crossover operator in evolutionary computing is to give individuals the ability to exchange parts of their genotype. In accord with selection pressure this should support the combination of high-quality sections of genotypes. In nature the crossover also brings resistance to errors caused by mutation of genotype. This is the reason why many researchers believe that a well-defined crossover is important, although approaches like evolutionary programming dealing with more complex genotypes usually abandon crossover in favor of sophisticated mutation operators.

The most common encoding of individuals in genetic programming (GP) represents a genotype as a tree structure. There are also other, mostly indirect, encoding approaches, such as evolving a set of commands describing creation process of the final tree, linear GP, or grammatical approaches. In this work we focus on a direct tree encoding where an individual is represented as a syntactic tree corresponding to the program it realizes.

There are several problems with applying crossover operator to tree structures describing the genotype. In many cases the best of ancestors produced from mating of two parent candidates is even worse than both of its parents. This is the reason why standard simple crossover (swapping randomly selected subtrees) [51] is often not the preferred operator in GP. The impact of operator applying is too strong in many cases, since a small change in individual's genotype tree structure can have a big and unpredictable impact to its phenotype.

The structure of this work is as follows: In the following section we briefly review relevant existing work on alternative crossovers for GP. Section 7.4.3 introduces various measures of subtree similarity. Next section introduces our environment for performing the tests, namely by describing details of a our genetic program. Results of experiments on standard benchmark tasks are reported in section 7.4.6 and discussed in the Conclusion section.

7.4.2 *Related work*

There have been several works on alternative crossover operators that improve the exchange of suitable subtrees. The context preserving crossover by D'Haeseleer [21] attempts to preserve the context in which subtrees appeared in the parent trees. The author introduced a coordinate scheme for nodes in a tree and allows crossover only between nodes with matching coordinates. Uy et al. [85] introduced a semantic distance of subtrees and defined a semantic similarity-based crossover which controls the distance of subtrees chosen for crossover. Their approach is tested on several symbolic regression problems.

A modification of crossover and mutation which measures performance of subtrees as a guidance for subtree selection has been proposed in [42]. Authors consider several approaches from simple performance of a subtree, to complexity and performance, to correlation of subtrees as the guidance factor. The monograph [9] also introduces a way how to measure structural similarity of subtrees utilized by crossover.

Beadle and Johnson in [10] introduce a semantically driven crossover which prevents offsprings to be semantically equivalent to their parents. The authors use a cannonical representation of trees in order to check the semantic equivalence without accessing the fitness. Their approach is further extended by [61] where authors investigate the effect of semantic guidance to the crossover. They define two operators, one considering the semantics of the exchanged subtrees, and one comparing the semantics of the offspring trees to their parents. They show that these operators perform better on a set of polynomial symbolic regression problems. A more detailed survey od semantically based methods of crossover operator in genetic programming is given by Vanneschi at al. [88].

The common element of all mentioned works is to prevent a distortion of a well performing subtree, which can be seem as a problem of breaking of a good building block. The idea behind our approach is very similar — to introduce only slight and reasonable changes in phenotype.

7.4.3 *Subtree similarity*

We present six methods to measure compatibility of two subtrees. These methods are compared with traditional crossover performing a random approach to find matching trees as a baseline. All methods take two (sub)-trees and compute a measure of similarity of them, based on various criteria.

In the following we consider these two example trees to clarify our methods (cf. Fig. 28):

$$T_a = \mathrm{add}(x, \mathrm{sqrt}(y))$$

$$T_b = \mathrm{sub}(x, 1)$$

The *average arity* method (AVA) computes average arity of functions in inner nodes. $\mathrm{AVA}(T_a) = (2+1)/2 = 1.5, \mathrm{AVA}(T_b) = 2$.

The *common variable set* method (CVS) counts the size of a set of the identical variables in the leaves of both subtrees. It takes the variables from leaves of T_a and from leaves of T_b. These two sets are thereafter intersected. The number of common variables then serves as a measure of semantic similarity of the subtrees. The crossover tends to choose more similar subtrees for exchange, i.e. those with more common variables. $\mathrm{CVS}(T_a) = \mathrm{CVS}(T_b)) = 1$ because the only one variable (x) is common for both trees.

The *common operator set* method does the same as *CVS* method, but instead of leaves' variables, it takes into account the functions (operators) from inner nodes. $COS(T_a) = COS(T_b) = \emptyset$

The *recurrent evaluation* method (REV) tries to represent a value of a subtree. It pushes a value of 0 to all variables in a tree, and retrieves a value from the root node. This value is again substituted to all variables, and so on for 20 times. The final root value is taken as a representative of given subtree. $\mathrm{REV}(T_a) = 0 + \sqrt{0} = 0, \mathrm{REV}(T_b) = -20$

The *vector evaluation* (VEV) method takes a randomly generated set of 20 numerical vectors $v_0, \dots, v_{20}$. The length of these vectors corresponds to the number of unique variables in compared trees – in our case of T_a and T_b it is 2 (for x and y). The tree is evaluated for each vector v_i, thus we obtain a vector r of 20 results for each of given trees T_a, T_b. Next, the *VEV* method computes the distance δ of these vectors by:

$$\delta(r^{T_a}, r^{T_b}) = \sum_{i=0}^{i<20} (r_i^{T_a} - r_i^{T_b})^2$$

The VEV method is similar to ideas introduced in semantic genetic programming [57] or semantic similarity-based crossover proposed by [85]. It measures the performance of the tree on given set of inputs.

The *string* (STR) method is based on a string serialization of tree. The given trees T_a, T_b are represented after serialization as S1 = "$\mathrm{add}(x, \mathrm{sqrt}(y))$" and S2 = "$\mathrm{sub}(x, 1)$". Next, the Levensthein distance [100] of these two strings is computed – $\mathrm{STR}(T_a, T_b) = 10$. This method should mirror the semantic equivalence of the trees.

For comparison, as a baseline solution, we also take randomly selected subtrees for crossover operator.

7.4.4 *Genetic programming test environment*

In the following we describe the problems to solve and the GP techniques we utilize as an environment to test our subtree similarity methods described above.

7.4.4.1 *The problem definition*

The problem at hand is the simple symbolic regression described in Koza [51]. The goal is to fit a real valued function by a tree evolved by the GP. The tree consists of inner nodes containing simple mathematical functions like addition, multiplication or abs(), and leaf nodes with numerical constants from $\mathbb{R}$, or variables. This variables also assume values from $\mathbb{R}$. The sample function is compared with evolved

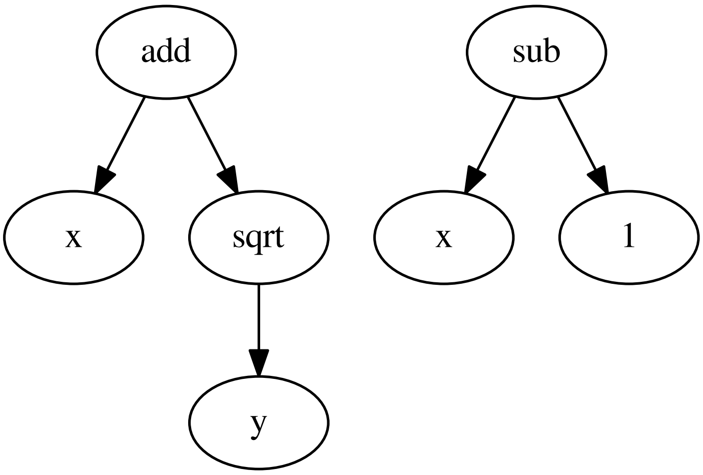


Figure 28: Two simple example trees.

function in defined amount of points from a given interval. Some datasets use equidistant grid of inputs and some use fixed number of uniform random samples drawn from given interval, as we describe in detail further.

For practical reasons we redefined some mathematical functions so they are defined in the whole interval $[-\infty, \infty]$. Also positive or negative ∞ was replaced by *MAX_DOUBLE* constant with matching sign during evaluation of a tree. Therefore:

- $\frac{x}{0} = 1$ for $x \in \mathbb{R}$
- $\frac{1}{x} = 1$ for $x = 0$
- $\log x = \log |x|$ for $x \in \mathbb{R} \setminus 0$, -20 for $x = 0$
- $\ln x = \ln |x|$ for $x \in \mathbb{R} \setminus 0$, -20 for $x = 0$
- $\sqrt{x} = \sqrt{|x|}$ for $x \in \mathbb{R}$

The set of available simple function consists of: $-, +, \times, \%, \sqrt{x}, |x|, 1/x, -x, x^2, \sin(x), \cos(x)$. The first four are binary functions (have an arity of 2) and the rest are unary functions (arity = 1). The % symbol represents a modulo operation.

7.4.4.2 *GP evolution configuration*

Each of the experiments was performed with configuration as shown in the table 6. Some specific variations are mentioned in particular experiments. The section 7.4.4.4 is dedicated to process of initialization of trees in individuals.

The fitness function f is defined as in equation 7.

$$f(t) = \sum_{i=0}^{N} |x(i) - t(i)| \tag{7}$$

Table 6: The configuration of GP in experiments.

population size	100
number of generations	200
selection method	tournament
tournament size	15
elitism	enabled
initialization	see Sec. 7.4.4.4
max. initial tree depth	10
operators	see Sec. 7.4.4.5

Where N is a number of sampling points of the fitted function x, $x(i)$ is value of the function in given point i and $t(i)$ is value returned by the evaluation of the tree which represents evaluated function. In all experiments the fitness function is maximized.

7.4.4.3 *Simple benchmark – Sample function sets*

This subsection describes how we prepared a simple benchmark set called SimpleF in our experiments. The benchmark is designed for two purposes. At first it is used to tune parameters of evolution, its initialization and selecting the best variant of adaptive operators approach. Secondary, it is used as one of several benchmarks to measure performance of proposed methods of subtree matching. The idea is to generate several random trees representing target functions, and to choose the medium complex ones as a set of fitness functions.

In a more detail, the target function which we try to fit, is a combination of simple functions mentioned above represented as tree. We generated 45 samples of full trees of depth 10 at random. These full trees are relaxed after initialization. The relaxation means that all subtrees which contain no variables are replaced by a constant leaf node. The deeper insight to initialization will be given in section 7.4.4.4.

For each from these trees we try to evolve a function which fits them by a standard GP procedure. The trees are afterward sorted by fitness function of the best evolved individual for each of them. Finally, the best and the worst thirds are removed, and the remaining 15 trees became a testing set. It can be presumed that functions represented by these trees are not so hard to fit for the GP. This dataset is named *SimpleF*.

7.4.4.4 *Tree initialization*

There are many initialization methods to construct a tree in GP [65]. Our sample function trees are full trees of a given depth. That means each leaf is in given depth and each inner node has number of children which matches to arity of function in that node. The function is selected from available simple functions at random. In leafs, there are randomly chosen constants from range $[0, 1]$ with uniform distribution. To ensure that the tree contains full set of variables we need to replace randomly selected leaf constants with variables. In the SimpleF dataset, there is only one variable x. In experiments on this dataset we need to use internally more labels for the same variable – eg. x_0, x_1, for the method *common variable set* (see section 7.4.3). Another benchmarks (see section 7.4.6) use more dimensional function samples which naturally imply labeling $x_0, \ldots, x_n$ for its variables and need no artificial labeling.

To achieve sufficient diversity in initial population in GP we combine several methods of tree initialization. The first method creates a minimal tree which contains all used variables. In the case of SimpleF dataset it means that tree consists of one inner node with arity one or two and matching number of child nodes. A such tree is a hopeful building block for construction of complex trees. The second and third methods are implementation of the grow, respectively the full approach, as described in [65]. Koza calls the combination of *grow* and *full* methods as *ramped half-and-half*. The last initialization method creates random tree with constants in leaves at given depth. In fact when this tree is relaxed (evaluated) it can be replaced by only one constant node – this may be a small disadvantage during the evolution when we want to avoid bloating of trees by relaxing them.

7.4.4.5 *Evolutionary operators*

There are two approaches how to apply GP operators to individuals. The operator can be applied to the selected individual only once per generation, and the newly created offsprings are preserved from other operators till the new generation is created. On the other hand, more operators can be applied in sequence to one individual. Usually the application of operators is given by the design of particular GP. The other possibility is to adaptively react to the performance of operators and favor the more successfull operators. This approach is known as self-adaptive evolution [20, 71].

In our approach, each operator O has its own probability $0 \leqslant P_O \leqslant 0.9$ to be applied. This probability is adaptively changed during evolution. When the ancestor has better fitness than the parent, the P_O of the applied operator is increased by 1×10^{-5}, if not, it is decreased by 1×10^{-6}. The sum of all P_O can exceed 1.0 but probabilities of all operators are then scaled to the range $[0, 1]$.

The following several mutation operators, and a crossover, guided by different similarity measured, were used in our experiments:

- *mutateLeafHillClimb*
  For a randomly selected leaf value the local space is searched, and the best found value replaces the old one.

- *mutateLeafByVal*
  A randomly selected leaf with constant value is changed by its momentum. This momentum is updated by -5 % to +5 %

- *mutateLeafAddSubtree*
  A randomly selected leaf is replaced by a newly generated tree of depth 3. The grow method for generating is used.

- *mutateLeafSwitchVariableConst*
  If selected leaf contains a constant, it is replaced by a randomly selected variable, and vice versa.

- *mutateInnerNodeFunction*
  In given inner node the function is changed to another with the same arity. When no function of current arity is available then a new function is chosen randomly and a proper count of child nodes are either constructed or destroyed as needed.

- *mutateInnerNodeToConstLeaf*
  The selected inner node is replaced by a constant and becomes a leaf.

- *cross*
  The cross operator take the second individual by tournament selection and tries to find a best matching subtrees of itself and the other tree. The measure of similarity will be defined in next section. The best matching subtrees are finally swapped.

7.4.5 *Experimental results of adaptive operators*

This section describes the experiments perfomed with different initialization of P_O's. There are four experiments as follows: *Uniformly distributed probabilities*, *preferred crossover*, *supressed crossover* and *only crossover*. All these experiments are performed on the *SimpleF* dataset.

7.4.5.1 *Uniformly distributed probabilities*

The first experiment put the same probability to all operators as you can see in table 7. The impact of adaptive changes in probabilities is shown as the second column. We can state that *leafHillclimb* and *leafSwitchVariableConst* operators brought better descendants then others. The probability of crossover operator left unchanged.

Table 7: Probabilities of operators before and after evolution. The values are averaged over 1000 runs with given setup.

operator	pre P_O	post P_O
mutateLeafHillClimb	0.143	0.228
mutateLeafByVal	0.143	0.174
mutateLeafAddSubtree	0.143	0.144
mutateLeafSwitchVariableConst	0.143	0.269
mutateInnerNodeFunction	0.143	0.144
mutateInnerNodeToConstLeaf	0.143	0.145
cross	0.143	0.143

Table 8: The crossover is strongly preferred operator but P_Os of mutation operators were increased in contrast to decreased P_O of crossover operator. The values are averaged over 1000 runs with given setup.

operator	pre P_O	post P_O
mutateLeafHillClimb	0.05	0.086
mutateLeafByVal	0.05	0.063
mutateLeafAddSubtree	0.05	0.051
mutateLeafSwitchVariableConst	0.05	0.104
mutateInnerNodeFunction	0.05	0.051
mutateInnerNodeToConstLeaf	0.05	0.052
cross	0.7	0.605

7.4.5.2 *Preferred crossover*

The second experiment examined the situation when the crossover has significantly higher P_O. The results are shown in the table 8. All mutation operators have increased their P_O. The crossover operator decreased its P_O by 13.6 %.This behavior brings idea of next experiment – the defended crossover.

7.4.5.3 *Suppressed crossover*

In this experiment there is blocked a punishment for crossover operator when it creates a worse ancestor then itself. The final P_Os of all mutation operators are nearly the same as in case of previous experiment – *preferred crossover*.

So we can state that using an evolution with more generations leads only to the primary reinforcement of two mutation operators. An only artificial defended crossover preserves its initial higher chance to be applied.

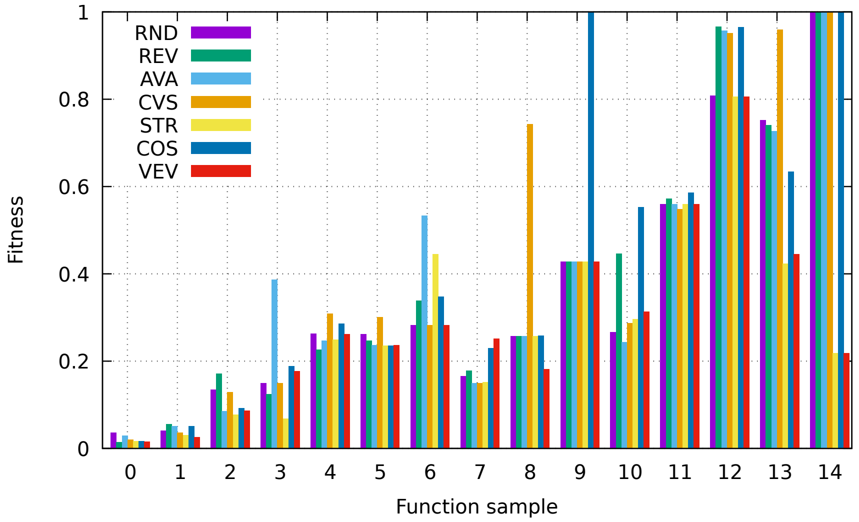


Figure 29: Averaged performance from 30 runs of each method on the SimpleF dataset.

7.4.5.4 *Crossover only*

In the last experiment of adaptive operators we have disabled all mutation operators. Therefore only the crossover operator is considered. The final crossover's P_O was 0.621 from initial 0.7. It can be stated that nevertheless the worse ancestors are still created by the crossover, the evolution is capable to find optimal solutions on SimpleF dataset. The figure 29 shows averaged performance from 30 runs of each method on the SimpleF dataset. Horizontal axis represents given samples which were fitted. On the vertical axis there is an average fitness function value of the best individuals.

For illustration, the best evolved trees from experiments on SimpleF dataset are presented on figures 30 and 31. Blue nodes represent variables, constants are gray, and yellow nodes represent atomic function.

For further experiments with subtree similarity methods we used *suppressed crossover* setup. The crossover had much higher probability ($P_O = 0.7$) then all other operators ($P_O = 0.05$ each). All other parameters of GP were the same as shown in the table 6.

7.4.6 *Experimental results of subtree matching*

The overall performance of the subtree matching methods are described in this section.

The survey [94] presented several GP benchmarks and discuss its suitability for GP experiments. Beside our own artificial dataset SimpleF for benchmarking (see Sec. 7.4.4.3) we adapted the following benchmark functions from the survey: kaijzer, nguyen7, pagie1, vladislavleva4. The last two show itself as too difficult – no one of pro-

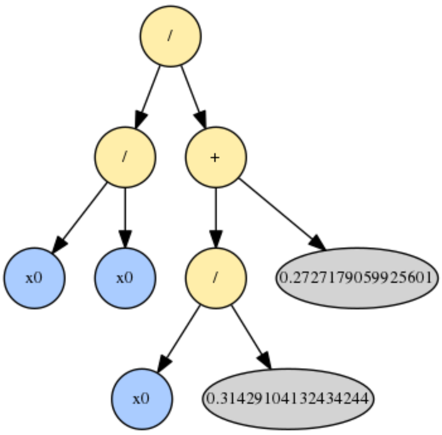


Figure 30: An example of the best evolved tree from experiments on SimpleF dataset.

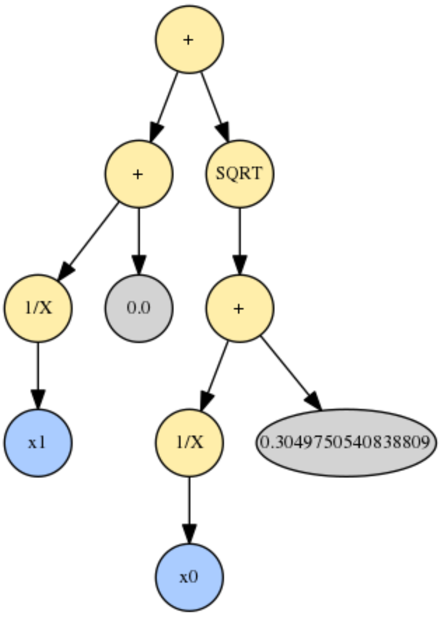


Figure 31: An example of the best evolved tree from experiments on SimpleF dataset. The tree is not relaxed during the evolution due to preservation of diversity in population – see additon of 0.0 in the left side of depicted tree.

posed methods was capable to perform significantly better than random baseline.

The table 9 shows benchmarks configuration. Each combination of matching method and dataset is computed 30 times to avoid an influence of a random essence of GP. Over all datasets totaly 61 unique function samples were fitted by each method.

Table 9: The used benchmarks configuration. U[a,b,c] is c uniform random samples drawn from a to b, inclusive, for the variable. E[a,b,c] is a grid of points evenly spaced (for this variable) with an interval of c, from a to b inclusive.

benchmark	# vars.	data points	simple functions
keijzer	1	E(1, 50, 1)	$+, \times, 1/n, -n, \sqrt{n}$
nguyen7	1	U(0, 2, 20)	$+, -, \times, /, \hat{}, \ln(n),$ $\sin(n), \cos n$
pagie1	2	E(-5, 5, 0.4)	$+, \times, 1/n, \hat{}$
vlad4	5	U(0.05, 6.05, 1024)	$+, -, \times, \%, n^2$
SimpleF	1	E(-5, 5, 20)	$+, \times, 1/n, -n, \sqrt{n}$

The results are summarized in the table 10. The bold text represents the best method in current experiment.

Each of methods was as good as the *random* baseline (RND) at least in one of benchmarks. We tested a null hypothesis as folows H_0: The means of samples of RND and another method on given dataset are the same. The results of T-test of H_0 are shown in table 11 – the table contains probabilities that the means of RND and given method are the same. In the table there is missing row for pagie1 dataset. It is caused by the fact that this dataset has only one function for fitting and therefore there are insufficient amount of data fot T-test. For keijzer dataset the *average arity* (AVA) method is significantly (at level 0.01) better than RND. On the same level of significance there is no another method better than RND baseline. On nguyen7 dataset there is *vector evaluation* (VEV) which is better than RND starting from significance level 0.1. The remaining two datasets vladislavleva4 and simpleF seems to be too difficult for all proposed methods and used configuration of GP.

7.4.7 *Conclusion*

The experiments with function fitting show that only *average arity* and *vector evaluation* crossovers outperformed the random matching of subtrees on specific dataset. The AVA method is about 15 % better on keijzer dataset than the RND baseline. The VEV outperformed RND on nguyen7. The remaining methods *recurrent evaluation*, *string*, *common operator ser* and *common variable set* are not significantly better

Table 10: The subtree similarity methods comparison on symbolic regression problems. The columns represent each of methods and their performance on given datasets. The results for each tuple method-dataset are averaged for all function samples in given dataset and 30 times repetition on each sample. The values are scaled by value or *RND* method.

	method						
dataset	**AVA**	**RND**	**CVS**	**STR**	**REV**	**COS**	**VEV**
keijzer	**1.150**	1.000	**1.040**	0.946	0.979	0.957	1.009
nguyen7	1.024	1.000	0.981	0.970	**1.003**	0.985	**1.041**
pagie1	0.997	1.000	0.976	1.013	0.980	0.991	0.998
vlad4	1.000	1.000	1.000	1.000	1.000	1.000	1.000
simpleF	1.059	1.000	0.971	**1.034**	0.995	**1.007**	0.867
avg	1.046	1.000	0.994	0.993	0.992	0.988	0.983
rank	1.	2.	3.	4.	5.	6.	7.

Table 11: The probabilities obtained from T-test for null hypothesis that means of RND and the another method are the same. The pagie1 dataset is missing because it has too small amount of data for statistical testing.

method	**AVA**	**CVS**	**STR**	**REV**	**COS**	**VEV**
keijzer	3,28E-06	0,18	0,06	0,01	0,06	0,38
nguyen7	0,24	0,47	0,26	0,18	0,31	0,09
vlad4	0,49	0,49	0,50	0,49	0,50	0,49
simpleF	0,46	0,50	0,48	0,48	0,50	0,41

than *random* approach to the crossover of trees on any dataset. It may be caused by the problem domain or the current setup – e.g. the initial constant supplied to REV method, or the selected text representation in STR method.

A REV method can be further modified – one can change a fixed number of iterations, make number of iterations variable, or stop when value converges to some stable value. An interesting idea is also to use another initial value for the first substitution – it may be made dependent on interval of desired function sample.

For the future work we will focus on the comparison of presented methods on another problem like the Lawnmower problem or Artificial Ant (both described in Koza [51]).

7.5 AUCTION SYSTEMS IN TASK DISTRIBUTION

The last section corresponds to the sub-problem proposed as follows: Evolve and compare control mechanisms of task distribution with task distribution driven by auctions systems. The best-known subtree matching method and the configuration of evolution parameters from previous experiments will be used in evolution. The experiment will be performed on real data mining tasks. The optimization criterion is a combination of mean error and average processing time for each task.

This journal article was published in the Revista Investigacion Operacional vol. 38, no. 4, in 2017.

Tree based Decision Strategies and Auctions in Computational Multi-agent Systems

Abstract: *This paper deals with an agent-based implementation of data mining system where a set of tasks is being processed in a distributed manner. The key role within such a system is the decision strategy of a computational agent which should consider accepting or rejecting a particular task based on various decision strategies. We present several adaptive decision strategies and compare them to traditional auction-based task distribution. Results show that optimal decision making strategy depends on the task set characteristic properties – e.g. how distinct are the best and the worst average results of each task type in dataset.*

7.5.1 *Introduction*

In the recent years, the data mining has become an area of rapid development both in theory and in applications. Modern computer systems in all fields of human interest are producing huge amounts of data. This data must be processed in reasonable time, and nowadays this is possible with a massive parallelization of processing. One suitable approach is known as *divide and conquer*. That means that one big task or task set is divided to smaller parts, these are solved separately, and the global result is constructed back from the smaller parts.

The data mining (DM) – sometimes called a *knowledge discovery* or *machine learning* – is the process of data analysis with an effort to obtain useful or valuable information, relationship or patterns which are not obviously evident from a raw data. Each intended result type needs another method. Let us focus on two areas of data mining: the *classification* and *regression*. In addition, for each of this areas there are dozens of algorithms and their variations and each of them is suitable for different data or in different circumstances.

The problem of data mining we are adressing is which method/algorithm is the most suitable one for a given data. The *metalearning* approach tries to answer this question. It is the subfield of machine learning which focuses on meta-data – the informations about data itself. The obtained knowledge of suitable algorithms and its configuration for given data brings better results such as higher precision or faster computation.

Basically, there are two possible approaches to task set distribution, one of them is the central control where one entity directs the splitting of a task set and rules the whole process. The central entity should know and take into account all additional knowledge about the task set which is being solved. In the other case, which can be called a *bottom-up* approach, the central entity divide the whole task set at random and the decision which part is suitable for which method lies on solving units. In this work we focused on the second approach.

The structure of this paper is as follows: The section 7.5.2 introduces the reader to the domain of multi-agent systems, later the principles of control of computation agent are discussed. The following chapter 7.5.3 is dedicated to the description of auction systems as the one of possible methods for task distribution. The following section 7.5.4 brings detailed overview of genetic programming used as the second option to develop control systems for task distribution. In sections 7.5.5 our experiment and results are described and the last chapter 8.1 sums up the whole work.

7.5.2 *Multi-agent systems for computation*

An agent is a computer system situated in an environment that is capable of autonomous action in this environment in order to meet its design objectives [93]. Its important features are adaptivity to changes in the environment and collaboration with other agents. Interacting agents join in more complex societies, *multi-agent systems* (MAS). These groups of agents gain several advantages such as the applications in distributed systems, delegacy of subproblems on other agents, and flexibility of the software system engineering. The *computational multi-agent systems*, i.e. application of agent technologies in the field of hybrid intelligence, showed to be promising by its flexibility and capability of parallel computation.

A *computational agent* is a highly encapsulated object realizing a particular computational method [60], such as a neural network, a genetic algorithm, or a fuzzy logic controller. Our system is designed to perform distributed data mining task solving.

Worker agents encapsulate data mining models and their goal is to solve tasks which were randomly picked instances of data sets and blindly distributed by *manager*. Each worker has to make a decision whether to accept or to reject the offered task to optimize designed criteria. The architecture we present can be easily modified or extended for application to different problems.

7.5.2.1 *Control of Computational Agent*

Two types of agents in our computational MAS are important when considering the agent adaptive control task; the *computational agent* whose role, as a worker is to accept or reject the offered tasks from the *manager*. Each accepted task should be solved with regard to optimization criteria. The manager distributes tasks to worker agents in a non-informed manner, i.e. it does not target specific task to specific agent, rather a random worker is selected from the pool of potentially available agents. A computational agent can accept only one task. If a worker is not actually processing a task it simply starts to work on the accepted task. On the other side, when it accepts a new task while solving another, it has to discard the task which is currently being solved. Such task is returned to the manager to be redistributed and the newly accepted task goes to processing.

The main goal of decision making is to accept tasks that are suitable for current working agent and to reject tasks which would be solved by this agent with worse value of optimization criterion – e.g. bigger error or greater elapsed time.

A worker agent should be able to make a decision which leads to acceptance of tasks which promise better results beeing solved by encapsulated model. It is also important to recognize whether to discard a started task which must be solved by another agent and take a new one instead of the discarded one with perspective to be solved better. This brings new requirements for our agents: an ability to compute tasks suitability for a model, case-based reasoning (see work of Aamodt [1]) to remember task cases which agent computed in the past, handling task's and evaluation's parameters (progress of task, elapsed time, system load, etc.).

We use following attributes as indicators of agent's state: *solvedTasks* – number of tasks solved by an agent, *expSolSteps* – expected steps to finish an offered task, *stepsSolved* – number of solved steps of currently processed task, and *currentSuit* – agent–task compatibility of currently solved task, *offeredSuit* – agent–task compatibility of offered task, *percentSSol* – computed percentage of the current task, and *ticksToEnd* – expected number of steps to finish actual task.

7.5.3 *Auction systems*

The auctions are well known principle how to reach agreements. There are two types of agents in auctions – the auctioneer and the bidders. In our case the *manager* is in the role of auctioneer and *workers* in roles of bidders. The goal of the auction is for the auctioneer to allocate resource to one of the bidders [98]. The resource is represented by an unsolved task from the task set.

Each *worker agent* has its own estimation *ew* of private value of offered task. The value of *ew* is based on statistical data about current data-mining model in *worker agent* and its performance metrics on the offered task. These attributes of a tuple agent-task are connected together by a polynomial expression. The proper formula is the subject of evolution by Genetic Programming – see section 7.5.4.

There are many types of bidding systems. Basic overview can be found at [25] in section 9.1. Now we focus only on two auction systems: *sealed* and *english*. Both of them are easily implementable in MAS.

7.5.3.1 *Sealed Auction*

In this kind of auction all bidders simultaneously send "sealed bids" with the offered price for resource to auctioneer. Then the auctioneer evaluate all of them simultaneously. The highest bid wins the resource and pay the offered price. The main pros is the minimal computation time for running this auction. Only three interactions are necessary: (1) a manager offers a task, (2) workers send their bids and (3) manager sends the task to the winner for processing.

7.5.3.2 *English Auction*

The English auction is also known as ascending-bid auction. Here the auctioneer in the real time announces the initial price for the resource. The bidders than can react with gradually rising offers. They drop out until the only one stay in and wins the resource for actual price. This system is more time consuming in comparison with the sealed auction as considerably higher number of rounds of interaction is required.

7.5.4 *Genetic Programming*

A genetic programming is an approach from family of evolutional algorithms where an individual is represented by a graph, more specifically by a tree. The common genetic algorithm is described in [51]. These trees are used for decision making whether to accept or to reject an offered task.

Similarly as proposed in [76] the tree represents a polynomial. The leaf elements of that tree contain attributes of the computational agent and also information about agent's environment such as the task currently being solved or a newly offered task. The domain of attribute values are integer or real numbers. Inner nodes of the tree represent operators with defined arity (e.g. ADD_2, SUB_2, MUL_2, NEG_1). Such tree is evaluated from leaf nodes to root (bottom-up).

The initial population is generated by modified ramped-half-and-half method [51]. The modification is as follows: part of all trees with fixed depth was not generated fully at random. There are constrains given by number of attributes. Each attribute should be at first weighted by a real number. It implies that nodes in level exactly one step above leaf nodes have always multiply operator (MUL). These nodes are also indifferent to mutation and crossover operator cannot split them. The leaf nodes with weights took their initial value at random with uniform distribution from $\langle 0, 1 \rangle$.

The behaviour of the mutation operator depends on the nature of the node – it is different for inner and leaf node. For the later one we have to consider also its value – whether it is a numerical value or a variable.

Numerical leaf nodes have their value slightly changed by application of mutation operator. This change preserves tendency of previous changes by use a momentum δ. This momentum is initialized as follows:

$$\delta = (-1)^r \cdot \frac{1}{3}\nu,$$

where r is chosen at random from set $\{0, 1\}$ and ν is the initial node value. After each application of mutation operator to a leaf node a node value is updated by addition of current value of δ. Immediately after that δ is updated by 20 % of its own value in a random way.

For faster convergence we tried to utilize hill climbing approach in the advanced mutation operator. This operator is applied only at leaf nodes with numerical values. It generates 10 new possible values of selected node in ϵ neighbourhood of the actual value in accordance with:

$$\nu_{T+1} = \nu_T \cdot (0.9 + r),$$

where ν_T is actual node value, ν_{T+1} is new node value and r is a random real number from range $\langle 0, 0.2 \rangle$. Then fitness function for the given individual is computed with each of new node values. The best fitness determines the final value of mutated node which will be used.

A mutation of leaf nodes with variable means that the present variable is replaced by another one from all available of agent's or environment's attributes. Genetic Programming behaves similarly in the case of mutation of inner nodes – another operator which has the same arity as the previous one is selected and set to the node.

The crossover operator takes trees from two individuals selected by tournament selection and in each of this trees it selects one node at random. These selected nodes with their subtrees are switched. We tried to prevent bloating of the tree by selecting the most compatible subtrees considering their depth and used attributes in leaves.

The application of each operator has probability $0.2 < p_o < 0.9$. These probabilities are initialized uniformly and therefore each operator has the same chance to be applied at the beginning of the evolution. However, during the evolution the probabilities are modified as follows: When an individual has a better fitness after application of the selected operator, this operator increases its p_o by 0.0001. The latter case – when the application of operator brings degradation of the fitness – means decrease of p_o by 0.00001. The difference in order of magnitude of the changes of p_o is designed by experimental results. The idea behind dynamic changes of p_o is that well working operator will be applied with higher probability.

The fitness function expresses quality of each individual. Commonly, it maps encoded form of an individual to a real number. In all experiments in this work we maximize the fitness value. Specific description of fitness computation will be described in section 7.5.5.

7.5.4.1 *Multipurpose General Expressions*

The elementary trees representing a polynomial expression by connecting all weighted attributes were replaced by a generalized model. This advanced form has added a conditional ternary operator *if* and brings more expressive power to our trees – see [30]. The *if* construct brings possibility to make several "smaller" decisions based only on some subset of attributes and combine them to the main result whether to accept or to reject the task.

With the *if* operator it is important to take care of the order of the child nodes during application of evolutionary operators. The first two child nodes are used as inputs for comparison, the third and the forth ones are *true* or *false* branch, respectively. We also tried to evolve trees with only *if* operator in inner nodes. This pure *if* trees model is commonly known as decision trees with top-down evaluation.

7.5.5 *Experiments*

The main goal of our experiment is to optimize the computation of a whole set of datamining (DM) tasks with respect to the selected criterion – task time, task error or both of them combined. For better performance of evolution of decision making systems the results for each pair model-task are precomputed. We used two datasets: At first we prepared artificial dataset where are utilized three DM models and five types of tasks. The artificial dataset has the following property: each of the three models is very good (reaches small error and shorter time) on the only one distinct type of task. Therefore we have two types of task which are difficult for all of DM models. This precondition showed as too strong.

The second dataset comes from OpenML repository [89]. We preselected *runs* (tuple: method-task-configuration in OpenML terminology) which has at least 100 results in repository. The need of filtering out the outliers from precomputed results was showed as very important. Compared with the artificial dataset mentioned before, the real data shows the fact that when some model is strong on a specific task it is also relatively strong on the majority of other task types.

7.5.5.1 *Fitness functions*

The fitness function was defined as follows:

$$f = 2 - E_t - T_t$$

where E_t is average error on the task t, T_t is average time needed for computation of one task from the task set. Both E_t and T_t are normalized to interval $\langle 0, 1 \rangle$. The normalization is necessary because each dataset has a different range of task's parameters. The numerical values of f are from $\langle 0, 2 \rangle$ and the fitness function was maximized during the evolution.

The evaluation of fitness function for each individual is very expensive in the terms of time. Calculation of each fitness needs to run the simulation and solving of a whole set of tasks. Thus we decided to use precomputed results of each considered pair of agent's inner model and task type. We took this precomputed results from our project Pikater [46] which is multi-agent data mining system.

7.5.5.2 *Results*

Performance of all methods was measured on two datasets consisting of 300 task in both cases.

The figure 32 shows the results for artificial dataset. The best result on task error criterion is achieved by a sealed auction with handmade equation for computing the offered price with averaged mean absolute error 0.3461. The same algorithm also gained the best results on time criterion with 2.1333 ticks per task on average.

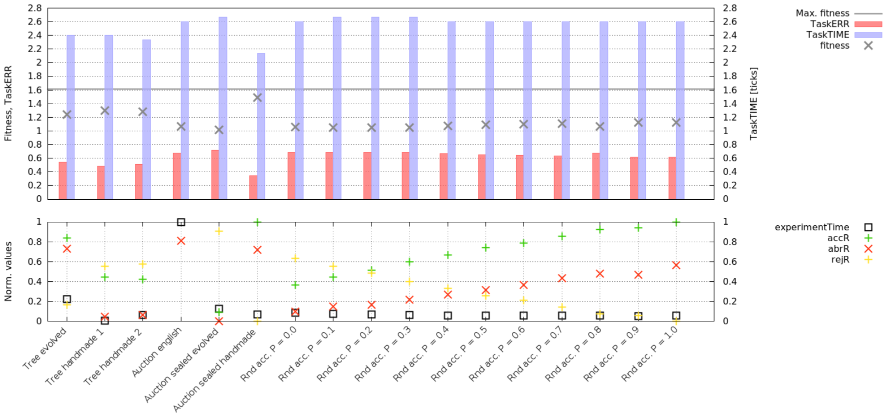


Figure 32: Detailed results of all decision making methods on Artificial data sets.

Table 12: Cumulative results of decision making methods on artificial data. The fitness and Mean Absolute Error (MAE) are unitless values and task time is given in ticks.

method	fitness		MAE		task time	
Random	1.0815	100.00 %	0.6600	100.00 %	2.6182	100.00 %
Tree	1.2743	**117.83 %**	0.5117	**77.52 %**	2.3778	**90.82 %**
Auction	1.1880	109.85 %	0.5808	88.00 %	2.4667	94.21 %

All 17 methods are divided into three groups – Random, Tree and Auction and results are averaged within each group. The random methods are used as a baseline. Either the decision making methods based on trees or auction systems overcame a random accepting of tasks in both criteria. The Cumulated results are shown in the table 12. The trees are around 17.83 % better than baseline and auction systems around 9.85 % better in comparison by fitness.

Figures 32 and 33 are divided into two parts. Both of them have a common horizontal axis with tested methods of decision making. The upper part shows fitness and its components – *task time* (measured in ticks) and *task error* (unitless value) and also maximal reachable fitness (unitless). The lower graph is aligned with the upper one and shows experiment time in seconds normalized to interval $\langle 0, 1 \rangle$. There are also plotted values of *task acceptance ratio* which is a ratio between count of all offered tasks to worker agents and tasks accepted by these agents, *task abortion ratio* which reflects how many task were aborted from all accepted tasks, and the *task rejection ratio* is complementary to task acceptance ratio and stands for the ratio between rejected tasks and all offered tasks.

On the OpenML dataset the same 17 methods were measured. Tree based methods performed approximately alike as random acceptance of the offered task. Auction systems outperformed random accep-

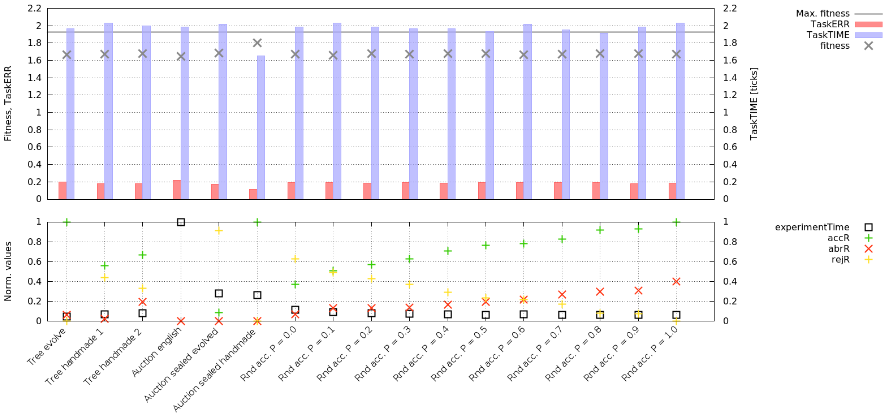


Figure 33: Detailed results of all decision making methods on OpenML data sets.

Table 13: Cumulative results of decision making methods on OpenML data. The fitness and Mean Absolute Error (MAE) are unitless values and task time is given in ticks.

method	fitness		MAE		task time	
Random	1.6734	100.00 %	0.1905	100.00 %	1.9788	100.00 %
Tree	1.6739	100.03 %	0.1868	98.11 %	2.0000	101.07 %
Auction	1.7119	**102.30 %**	0.1677	**88.04 %**	1.8833	**95.18 %**

tance method only by 2.3 % on average. See figure 33 for detailed results on OpenML datasets.

7.5.6 *Conclusion*

We proposed several types of decision making systems for task accepting problem in computational multi-agent systems. As a baseline we use random decision making with different probability of accepting offered task. The second approach is based on two auction systems – an english auction and a sealed auction. The last group of methods builds up on tree structures.

Experimental data sets are of two types – the first one is artificially made dataset with specific properties (see sec. 7.5.5.2) and the second one is taken from OpenML repository. All results of datamining task are precomputed for evolution and experiment speed up.

On the artificial dataset there is the best result achieved by sealed auction with hand made estimation formulae; however, on average the tree based methods are better – approximately 17.8 % above baseline. The results on OpenML dataset are more tight – auction systems overcame baseline only by 2.3 %. The reason of tight results on OpenML dataset is caused by the fact that when some datamining model is good on some data then it is also comparatively good on other data.

The future work may focus on further investigation of matching of pair: computation model–task. A promising approach is shown in [105] where authors are using meta-data information derived automatically from a data set to introduce the notion of similarity on data sets, and then training a recommending model to optimize a data set–model assignment. Also, another real-world data will by used in experiments.

Part IV

CONCLUSION

8

CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSIONS

The possibilities of adaptive control of computational agents solving data mining problems were studied. The Genetic Programming (GP) as a method of automated evolution of tree structures was used. Both automatically evolved and human-designed tree structures were compared as decision-making systems encapsulated in our agents. The agent's behaviors were controlled by these adaptive decision-making systems.

An experimental environment was defined as follows. We have a set of data mining tasks. Each task consists of data itself and metadata which contain additional information about data. The whole task set should be solved concerning defined criterion (e. g. the smallest mean squared error or the shortest time). On the other side, we have a set of computational agents which encapsulate some of the data mining models for classification, regression, etc. Another type of agent – the manager distributes tasks from the task set to computational agents and collects finalized tasks back. This manager does not use any prior knowledge about the suitability of the given task and randomly selected computational agent. The responsibility lies on the side of the computational agent and its decision making process during acceptance of the offered task.

The process of automatic evolution of decision-making systems by GP was studied. We focused on advanced genetic operators. Especially, to find an efficient and reliable crossover operator is a challenge due to its impact on the semantic of subtrees. A small change in tree structure may dramatically change the semantic of the whole tree. In the words of evolutionary algorithms, a small change in genotype could bring a huge change in phenotype. One of the possible ways how to face this problem is to try to make a relatively small change in the structure of an individual. The basic concept of a crossover operator is to swap subtrees in the parent individuals. Therefore we want to switch the most similar but still distinct subtrees from parents. The several methods for subtree similarity matching were proposed.

According to partial goals proposed in the section 1.2, we started with verifying the base concept of distributed tasks solving by com-

Goal 1: Explore the possibilities to control agents in MAS by decision trees in simple experimental setup.

putational agents on artificial datasets. We created five classes of tasks and defined the specialization of each computational agent. The time measured in ticks was used as the optimization criterion. The tree structures connecting different attributes of agent and task were evolved by GP. These trees were evaluated as polynomial and compared to a defined threshold during the decision-making process when a task was offered from the manager to the computational agent. As a baseline, the random task accepting method with a different probability of acceptance of the offered task was used. The results show that best-evolved expression tends to achieve the shortest queue of tasks in computational agent's incoming buffer. It is caused by the fact that the waiting time of a task is counted into the total time spent on a particular task. The best solution was approximately 12 times better than random baseline with 50 % probability of acceptance.

Goal 2: Investigate the feasibility to use GP evolved trees on complex tasks.

The next step was naturally to use real data and data mining models. Several datasets from the UCI Machine learning repository were used as benchmarking data. The computational agents encapsulated models like Multilayer perceptron, Radial-basis function network or Naive Bayes. The main criterion of fitness function was the averaged mean squared error achieved on the whole task set. The rest of the experiment setup remained the same as before. The results show that the best-evolved trees achieved averaged task error better by 33.5 % than baseline results. The total experiment time was approximately three times shorter because the computational agents learned to accept the tasks which were suitable for them.

The experiments have shown that crossover operator can bring better individuals, but the results are strongly affected by random aspect – the good trees were often destructively modified by crossover. Our expectations regarding the impact of the crossover operator were confirmed and the legitimacy of our 4th goal was supported. These cases guided later our steps to the investigation of the methods of crossover and tree matching.

Goal 3: Extend the SCO to use multiple criteria.

Our following steps went to the combination of both previously used optimization criteria. The two approaches were proposed – a linear combination of time and error criterion and multi-criteria optimization of both criteria at once. As a multi-criteria optimization method, the NSGA-II algorithm was used. A few new agent's attributes were added, and the incoming buffer of computation agent was reduced to the length 1. The shortening of the agent's buffer brought more responsibility to an agent's decision making. The abortion of the task currently being solved (by accepting the newly offered one) became much more expensive because an agent could not store it in its buffer and had to send it back to the manager agent as unsolved. The second major change was the newly introduced inner node function *if* which brings more expressive power of our trees.

An influence of newly added inner node operator *if* was manifested only when a linear combination of time and error criterion was used. When only one simple criterion (time or error) was used, the results were slightly worse with the utilization of *if*. The conclusion is that a more complex structure is better for more complex fitness function.

The MCO provided comparable results in the error criterion; however, in the time criterion, the results were below average. The problem with time criterion was solved by providing additional derived attributes to datasets. After this improvement, the best individuals from MCO achieved comparable results with the single criterion approach even in the time domain.

Goal 4: Explore the possibilities and propose an improvement of crossover operator in GP.

As stated above, the crossover operator needed special attention during the evolution of trees in GP. We studied techniques that should reduce the destructive impact of a crossover operator. Because a small change in genotype can cause a great change in phenotype, we focused on the reduction of these changes. Smaller changes are achievable when the crossed subtrees have similar semantics. Therefore we proposed several methods of subtree matching and compared their performance in the evolution process. The symbolic regression on several datasets was used as a benchmark. Only the *average arity* method outperformed a selected baseline significantly. All other methods managed to be better than baseline only on specific datasets.

Goal 5: Compare tree-based control mechanisms with another task distribution control system.

The process of task distribution was designed from the manager agent's point of view as random. The responsibility for effective distribution of tasks lied on a computational agent's decision making. This approach was named as bottom-up decision making architecture. We compared our approach to task distribution with the idea to use the auction system (Sealed and English auction) for task distribution. As testing datasets, we used two types of data. The first were artificially generated data and the second were well-known tasks from OpenML library. On artificial datasets, both – the tree based decision-making systems and the auction systems – overcame baseline approximately by 13.8 %. Tree-based methods were slightly better than auction systems. On the real task, the results were tighter. Auction systems overcame baseline only by 2.3 %. The tight results on OpenML dataset were caused by the fact that when some data mining model is performing well on some data, then it is also comparatively good on other data. We assume that a substantial role on the mentioned results is also played by some similarities in the datasets from OpenML.

The capability of GP to evolve adaptive control mechanisms for the problem of *bottom-up* oriented decision making in task allocation in a data-mining MAS was studied. Based on performed experiments and our research we can state that the idea of bottom-up oriented decision making during the computational process in data mining MAS seems to be viable. The proposed method *average arity* for subtree matching in crossover operator outperformed other proposed and existing methods.

8.2 FUTURE WORK

Several interesting areas were explored during various experiments. Further research can be focused on testing the proposed approaches on more datasets from different domains or on the real world problems. The application of evolutionary approaches to the real world problems brings challenging questions such as: How to deal with the extremely expensive evaluation of fitness function when no precomputed results can be used? How to select the best available individual encoding?

The selection of the agent's and the task's attributes can be systematically explored. We prepared the set of attributes by hand. The evolution process picked up some of them and preferred them to others by different weights. The process of initial selection of attributes can also be automated. It may be a field for the application of metaevolution.

A big chapter is the role of crossover operator in GP. Taking care about semantics similarity of subtrees makes sense although the definition of such similarity measures seems to be demanding and deserves more investigation.

BIBLIOGRAPHY

[1] Agnar Aamodt. "Explanation-Driven Case-Based Reasoning." In: *Topics in case-based reasoning*. Springer-Verlag, 1994, pp. 274–288.

[2] Agnar Aamodt and Enric Plaza. "Case-based reasoning : Foundational issues, methodological variations, and system approaches." In: *AICom — Artificial Intelligence Communications* 7.1 (1994), pp. 39–59. URL: `citeseer.nj.nec.com/252387.html`.

[3] Forest Agostinelli, Guillaume Hocquet, Sameer Singh, and Pierre Baldi. "From Reinforcement Learning to Deep Reinforcement Learning: An Overview." In: *Braverman Readings in Machine Learning. Key Ideas from Inception to Current State*. Springer, 2018, pp. 298–328.

[4] David Aha and Dietrich Wettschereck. "Case-based learning: Beyond classification of feature vectors." In: *Machine Learning: ECML-97* (1997), pp. 327–336.

[5] David P. Anderson, Eric Korpela, and Rom Walton. "High-performance task distribution for volunteer computing." In: *First International Conference on e-Science and Grid Computing (e-Science'05)*. IEEE. 2005, 8–pp.

[6] F. Baader et al. *The description logic handbook: Theory, implementation, and applications*. Cambridge University Press, 2003.

[7] K. Bache and M. Lichman. *UCI Machine Learning Repository*. 2013. URL: `http://archive.ics.uci.edu/ml`.

[8] *Bang3 Web page*. [cit. 2011–06–15]. URL: `http://www.cs.cas.cz/bang3/`.

[9] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1-55860-510-X.

[10] Lawrence Beadle and Colin G Johnson. "Semantically driven crossover in genetic programming." In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE. 2008, pp. 111–116.

[11] Sean Bechhofer. "OWL: Web ontology language." In: *Encyclopedia of Database Systems*. Springer, 2009, pp. 2008–2009.

[12] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*. Vol. 7. John Wiley & Sons, 2007.

[13] Tobias Blickle and Lothar Thiele. “Genetic programming and redundancy.” In: *Choice* 1000 (1994), p. 2.

[14] Lashon B. Booker, David E. Goldberg, and John H. Holland. “Classifier systems and genetic algorithms.” In: *Artificial intelligence* 40.1 (1989), pp. 235–282.

[15] Pavel Brazdil, Christophe Giraud Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning: Applications to data mining*. Springer Science & Business Media, 2008.

[16] Longbing Cao, Vladimir Gorodetsky, and Pericles A Mitkas. “Agent mining: The synergy of agents and data mining.” In: *IEEE Intelligent Systems* 24.3 (2009).

[17] Soumen Chakrabarti, Martin Ester, Usama Fayyad, Johannes Gehrke, Jiawei Han, Shinichi Morishita, Gregory Piatetsky-Shapiro, and Wei Wang. *Data Mining Curriculum: A Proposal*. Tech. rep. "ACM SIGKDD", 2006.

[18] Bertrand Clarke, Ernest Fokoue, and Hao Helen Zhang. *Principles and theory for data mining and machine learning*. Springer Science & Business Media, 2009.

[19] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. “A fast and elitist multiobjective genetic algorithm: NSGA-II.” In: *Evolutionary Computation, IEEE Transactions on* 6.2 (2002), pp. 182–197. ISSN: 1089-778X. DOI: 10.1109/4235.996017.

[20] Kalyanmoy Deb and Hans-Georg Beyer. “Self-adaptation in real-parameter genetic algorithms with simulated binary crossover.” In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*. Morgan Kaufmann Publishers Inc. 1999, pp. 172–179.

[21] P. D’haeseleer. “Context preserving crossover in genetic programming.” In: *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. 1994, 256–261 vol.1. DOI: 10.1109/ICEC.1994.350006.

[22] Claudia Diamantini, Domenico Potena, and Emanuele Storti. “KDDONTO: An Ontology for Discovery and Composition of KDD Algorithms.” In: *ECML/PKDD09 Workshop on Third Generation Data Mining: Towards Service-oriented Knowledge Discovery*. 2009, pp. 13–24.

[23] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2010. ISBN: 3642142788.

[24] Edmund H Durfee and Victor R Lesser. "Negotiating task decomposition and allocation using partial global planning." In: *Distributed artificial intelligence* 2.1 (1989), pp. 229–244.

[25] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[26] Russell C Eberhart and Yuhui Shi. *Computational intelligence*. Morgan Kaufmann, 2007.

[27] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Vol. 53. Springer, 2003.

[28] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[29] *FIPA ACL Message Structure Specification*. http://www.fipa.org/.

[30] Matthias Felleisen. "On the Expressive Power of Programming Languages." In: *Science of Computer Programming*. Springer-Verlag, 1990, pp. 134–151.

[31] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. "KQML as an agent communication language." In: *Proceedings of the third international conference on Information and knowledge management*. ACM. 1994, pp. 456–463.

[32] ACL Fipa. "Fipa acl message structure specification." In: *Foundation for Intelligent Physical Agents* (2002). URL: `http://www.fipa.org/specs/fipa00061/SC00061G.html` (visited on 06/30/2004).

[33] Chris Giannella, Ruchita Bhargava, and Hillol Kargupta. "Multi-agent systems and distributed data mining." In: *International Workshop on Cooperative Information Agents*. Springer. 2004, pp. 1–15.

[34] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms." In: *Foundations of genetic algorithms* 1 (1991), pp. 69–93.

[35] David E Goldberg and Jon Richardson. "Genetic algorithms with sharing for multimodal function optimization." In: *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum. 1987, pp. 41–49.

[36] Vladimir Gorodetsky, Oleg Karsaeyv, and Vladimir Samoilov. "Multi-agent technology for distributed data mining and classification." In: *Intelligent Agent Technology, 2003. IAT 2003. IEEE/WIC International Conference on*. IEEE. 2003, pp. 438–441.

[37] Mark A Hall and Geoffrey Holmes. "Benchmarking attribute selection techniques for discrete class data mining." In: *IEEE Transactions on Knowledge and Data engineering* 15.6 (2003), pp. 1437–1447.

[38] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[39] John H Holland. "Genetic algorithms." In: *Scientific american* 267.1 (1992), pp. 66–72.

[40] M.N. Huhns and M.P. Singh. *Readings in Agents*. Morgan Kaufmann, 1998. ISBN: 9781558604957. URL: http://www.google.cz/books?id=w83nIszG9-YC.

[41] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004. ISBN: 052154310X.

[42] Hitoshi Iba and Hugo de Garis. "Extending genetic programming with recombinative guidance." In: *Advances in genetic programming* 2 (1996), pp. 69–88.

[43] *JADE (Java Agent DEvelopment Framework)*. 2011. URL: http://jade.tilab.com/.

[44] Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. "A roadmap of agent research and development." In: *Autonomous agents and multi-agent systems* 1.1 (1998), pp. 7–38.

[45] Paul Judas and Lorraine Prokop. "A historical compilation of software metrics with applicability to NASA's Orion spacecraft flight software sizing." In: *Innovations in Systems and Software Engineering* 7 (3 2011). 10.1007/s11334-011-0142-7, pp. 161–170. ISSN: 1614-5046. URL: http://dx.doi.org/10.1007/s11334-011-0142-7.

[46] Ondřej Kazík, Klára Pešková, Martin Pilát, and Roman Neruda. "Meta Learning in Multi-agent Systems for Data Mining." In: *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on* 2 (2011), pp. 433–434. DOI: http://doi.ieeecomputersociety.org/10.1109/WI-IAT.2011.233.

[47] Ondřej Kazík. "Adaptive Matchmaking Algorithms for Computational Multi-Agent Systems." PhD thesis. Charles University in Prague, 2014.

[48] Antonin Komenda, Jiri Vokrinek, Michal Cap, and Michal Pechoucek. "Developing Multiagent Algorithms for Tactical Missions Using Simulation." In: *IEEE Intelligent Systems* 28(1).1 (Jan. 2013), pp. 42–49. URL: http://doi.ieeecomputersociety.org/10.1109/MIS.2012.90.

[49] A. J. Kornecki and J. Zalewski. "Software Development for Real-Time Safety-Critical Applications." In: *Software Engineering Workshop - Tutorial Notes, 2005. 29th Annual IEEE/NASA*. Apr. 2005, pp. 1 –95. DOI: 10.1109/SEW.2005.6.

[50] John R Koza. "Genetic evolution and co-evolution of computer programs." In: *Artificial life II* 10 (1991), pp. 603–629.

[51] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, 1992. ISBN: 9780262111706.

[52] John R Koza. "Genetic programming II: Automatic discovery of reusable subprograms." In: *Cambridge, MA, USA* (1994).

[53] R. Kraft, Q. Lu, and M. Wisebond. *Task distribution processing system and the method for subscribing computers to perform computing tasks during idle time.* US Patent 6,112,225. Aug. 2000. URL: `https://www.google.com/patents/US6112225`.

[54] William B Langdon. "Scaling of program fitness spaces." In: *Evolutionary Computation* 7.4 (1999), pp. 399–428.

[55] William B Langdon. "Quadratic bloat in genetic programming." In: *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2000, pp. 451–458.

[56] William B Langdon and Wolfgang Banzhaf. "Genetic programming bloat without semantics." In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2000, pp. 201–210.

[57] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. "Geometric semantic genetic programming." In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2012, pp. 21–31.

[58] Richard Murdoch and Tony Johnson. *Intelligent software agents*. Prentice Hall PTR, 1999.

[59] Roman Neruda and Gerd Beuster. "Emerging Hybrid Computational Models." In: *Proc. of the ICIC 2006*. .2. LNCS 4113. 2006, pp. 379–389.

[60] Roman Neruda, Pavel Krušina, and Zuzana Petrova? "Towards soft computing agents." In: *Neural Network World* 10.5 (2000), pp. 859–868.

[61] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. "Semantic aware crossover for genetic programming: the case for real-valued function regression." In: *European Conference on Genetic Programming*. Springer. 2009, pp. 292–302.

[62] Peter Novák. "Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations." In: *International Workshop on Programming Multi-Agent Systems*. Springer. 2008, pp. 72–87.

[63] Natalya F Noy, Deborah L McGuinness, et al. *Ontology development 101: A guide to creating your first ontology*. 2001.

[64] H Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. "A design taxonomy of multi-agent interactions." In: *International Workshop on Agent-Oriented Software Engineering*. Springer. 2003, pp. 123–137.

[65] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. (With contributions by J. R. Koza). 2008. URL: `http://www.gp-field-guide.org.uk`.

[66] Miroslav Prýmek. "Multiagentní systémy v praxi." MA thesis. Masarykova univerzita, Brno, 2008.

[67] A. S. Rao and M. P. Georgeff. "BDI-agents: from theory to practice." In: *Proceedings of the First Intl. Conference on Multiagent Systems*. San Francisco, 1995. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.9247`.

[68] Anand S Rao. "AgentSpeak (L): BDI agents speak out in a logical computable language." In: *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Springer. 1996, pp. 42–55.

[69] Anatol Rapoport and Albert M Chammah. *Prisoner's dilemma: A study in conflict and cooperation*. Vol. 165. University of Michigan press, 1965.

[70] Lior Rokach and Oded Maimon. *Data mining with decision trees: theory and applications*. World scientific, 2014.

[71] N Saravanan, David B Fogel, and Kevin M Nelson. "A comparison of methods for self-adaptation in evolutionary algorithms." In: *BioSystems* 36.2 (1995), pp. 157–166.

[72] David Sislak, Premysl Volf, Dusan Pavlicek, and Michal Pechoucek. "AGENTFLY: Multi-Agent Simulation of Air-Traffic Management." In: *20th European Conference on Artificial Intelligence*. Montpellier, France: IOS Press, 2012. ISBN: 978-1-61499-097-0.

[73] David Sislak, Premysl Volf, Michal Pechoucek, Christopher T. Cannon, Duc N. Nguyen, and William C. Regli. "Multi-Agent Simulation of En-Route Human Air-Traffic Controller." In: *Proceedings of the Twenty-Fourth Innovative Appications of Artificial Intelligence Conference*. Toronto, Canada: AAAI Press, 2012, pp. 2323–2328. ISBN: 978-1-57735-568-7.

[74] Martin Šlapák. "Genetics in decision behaviour of computational agents." In: *Proceedings of Mendel 2011 - 17th International Conference on Soft Computing*. 2011.

[75] Martin Šlapák and Roman Neruda. "Evolving Decision Strategies for Computational Intelligence Agents." In: *Proceedings of ICIC 2012 - Lecture Notes in Artificial Inteligence*. 2012.

[76] Martin Šlapák and Roman Neruda. "Multiobjective Genetic Programming of Agent Decision Strategies." In: *Proceedings of the Fifth International Conference on Innovations in Bio-Inspired Computing and Applications IBICA 2014*. Ed. by Pavel Kömer, Ajith Abraham, and Václav Snášel. Vol. 303. Advances in Intelligent Systems and Computing. Springer International Publishing, 2014, pp. 173–182. ISBN: 978-3-319-08155-7. DOI: `10.1007/978-3-319-08156-4_18`.

[77] Terence Soule. "Code growth in genetic programming." PhD thesis. University of Idaho, 1998.

[78] Terence Soule and Robert B Heckendorn. "An analysis of the causes of code growth in genetic programming." In: *Genetic Programming and Evolvable Machines* 3.3 (2002), pp. 283–309.

[79] Peter Stone and Manuela Veloso. "Multiagent systems: A survey from a machine learning perspective." In: *Autonomous Robots* 8.3 (2000), pp. 345–383.

[80] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[81] Zbigniew Michalewicz Thomas Bäeck David B. Fogel. *Evolutionary Computation 2. Taylor & Francis. 2000.* Taylor & Francis, 2000. ISBN: 0750306653.

[82] Sarah Rebecca Thomas. "PLACA, an agent oriented programming language." 1993.

[83] Alejandro Torreño, Eva Onaindia, Antonín Komenda, and Michal Štolba. "Cooperative multi-agent planning: a survey." In: *ACM Computing Surveys (CSUR)* 50.6 (2018), p. 84.

[84] Padhraic Smyth Usama Fayyad Gregory Piatetsky-Shapiro. *From Data Mining to Knowledge Discovery in Databases*. `http://www.kdnuggets.com/gpspubs/aimag-kdd-overview-1996-Fayyad.pdf`, [cit: 2012-05-10]. 1996.

[85] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I McKay, and Edgar Galván-López. "Semantically-based crossover in genetic programming: application to real-valued symbolic regression." In: *Genetic Programming and Evolvable Machines* 12.2 (2011), pp. 91–119.

[86] Roman Vaculin. "Artificial intelligence models in adaptive agents." MA thesis. Prague: Faculty of Mathematics and Physics, Charles University, 2003.

[87] Roman Vaculin and Roman Neruda. *Concept nodes architecture within the Bang3 system*. Tech. rep. Institute of Computer Science, Academy of Science of the Czech Republic, 2004.

[88] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. "A survey of semantic methods in genetic programming." In: *Genetic Programming and Evolvable Machines* 15.2 (2014), pp. 195–214.

[89] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. "OpenML: Networked Science in Machine Learning." In: *SIGKDD Explorations* 15.2 (2013), pp. 49–60. DOI: 10.1145/2641190.2641198. URL: http://doi.acm.org/10.1145/2641190.2641198.

[90] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*. Vol. 1. Wiley New York, 1998.

[91] José M. Vidal. *Fundamentals of Multiagent Systems: Using NetLogo Models*. http://www.multiagent.com. Unpublished, 2006. URL: http://www.multiagent.com/fmas.

[92] Mathijs Weerdt and Brad Clement. "Introduction to planning in multiagent systems." In: *Multiagent and Grid Systems* 5 (Dec. 2009), pp. 345–355. DOI: 10.3233/MGS-2009-0133.

[93] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[94] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. "Better GP benchmarks: community survey results and proposals." In: *Genetic Programming and Evolvable Machines* 14.1 (2013), pp. 3–29. ISSN: 1573-7632. DOI: 10.1007/s10710-012-9177-2. URL: http://dx.doi.org/10.1007/s10710-012-9177-2.

[95] Darrell Whitley. "A genetic algorithm tutorial." In: *Statistics and Computing* 4 (2 1994). 10.1007/BF00175354, pp. 65–85. ISSN: 0960-3174. URL: http://dx.doi.org/10.1007/BF00175354.

[96] M. Wooldridge. *Intelligent Agents*. 1999.

[97] M. Wooldridge, N. R. Jennings, and D. Kinny. "The Gaia Methodology for Agent-Oriented Analysis and Design." In: *Journal of Autonomous Agents and Multi-Agent Systems* 3.3 (2000), pp. 285–312.

[98] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[99] Michael Wooldridge and Nicholas R Jennings. "Agent theories, architectures, and languages: a survey." In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1994, pp. 1–39.

[100] Li Yujian and Liu Bo. "A normalized Levenshtein distance metric." In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095.

[101] Mohammed J Zaki. "Parallel and distributed association mining: A survey." In: *IEEE concurrency* 7.4 (1999), pp. 14–25.

[102] Ivan Zelinka. *Symbolic regression – an overview*. online. [cit. 2017–07–15] `http://www.mafy.lut.fi/EcmiNL/older/ecmi35/node70.html`. 2009.

[103] Li Zeng, Ling Li, Lian Duan, Kevin Lu, Zhongzhi Shi, Maoguang Wang, Wenjuan Wu, and Ping Luo. "Distributed data mining: a survey." In: *Information Technology and Management* 13.4 (2012), pp. 403–409.

[104] Z. Zhang and Ch. Zhang. *Agent-Based Hybrid Intelligent Systems*. Springer Verlag, 2004.

[105] Jakub Šmíd and Roman Neruda. "Using Genetic Programming to Estimate Performance of Computational Intelligence Models." English. In: *Adaptive and Natural Computing Algorithms*. Ed. by Marco Tomassini, Alberto Antonioni, Fabio Daolio, and Pierre Buesser. Vol. 7824. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 169–178. ISBN: 978-3-642-37212-4. DOI: `10.1007/978-3-642-37213-1\_18`. URL: `http://dx.doi.org/10.1007/978-3-642-37213-1\_18`.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LYX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`