Computing and Informatics, Vol. 32, 2013, 1055–1078

AGENT STRATEGY GENERATION BY RULE INDUCTION

Bartłomiej Śnieżyński

AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: Bartlomiej.Sniezynski@agh.edu.pl

Abstract. This paper presents a study on a rule induction application for generating an agent strategy. It is a new approach in multi-agent systems, where reinforcement learning and evolutionary computation is broadly used for this purpose. Experimental results show that rule induction improves agent performance very quickly. What is more, rule-based knowledge representation has many advantages. It is comprehensive and clear. It allows for the examination of the learned knowledge by humans. Because of modularity of the knowledge, it also allows for the implementation of the knowledge exchange in a natural way – only necessary set of rules can be sent. Rule induction is tested in two domains: Fish Banks game, in which agents run fishing companies and learn how to allocate ships, and Predator-Prey domain, in which predator agents learn how to capture preys. The proposed learning mechanism should be beneficial in all domains, in which agents can determine the results of their actions.

Keywords: Multi-agent systems, rule induction, machine learning

1 INTRODUCTION

Decentralized problem solving is a method to deal with complexity. One of the architectures, which can be used for such a purpose is multi-agent system. In complex or changing environments it is very difficult, sometimes even impossible, to design all system details a priori. To overcome this problem one can apply a learning algorithm, which allows for the adoption of the system to the environment.

To use machine learning techniques in a multi-agent system, one should choose a specific method of learning, which fits well to the problem. There are many

algorithms developed so far. However, in multi-agent systems most applications use reinforcement learning or evolutionary computations. The goal of our research is to check if another method – rule induction – can also be used for strategy generation.

For reinforcement learning feedback from the environment rating the last action executed is sufficient to learn the strategy. A similar situation is for evolutionary computation. In the contrast, rule induction is a supervised learning method; therefore, it needs training data in a form of labeled examples to generate the knowledge.

Fortunately, the learning agent is able to generate training data using its experience. The condition is that the agent should be able to determine the impact of the action chosen for execution in a given state to its goal(s) and determine which actions are good and which are bad. Actions which have good performance in a given state are used as positive examples, and actions with poor performance as negative ones. If the results of actions performed by the agent are visible immediately, and the agent is able to determine their impact, the situation is clear. In such cases inductive learning can be applied directly. It may be the case in relatively simple environments, like ones tested in this research. Also, it may be used by a complex agent for generating strategy for a part of its responsibility for which it is applicable. This defines the class of problems in which the agent is able to use rule induction to directly define its strategy in a way described here.

Agent strategy can be generated in a similar way not only by using rule induction but also using any supervised learning method. However, rule-based knowledge representation has several advantages, especially if rules are unordered. Rules seem to correspond to a human way of thinking very well [16, 14]. Therefore, it is possible to understand and verify generated knowledge. Modularity of this knowledge representation allows for an exchange of the knowledge in a simple way. If an agent has no idea what to do in a given situation, it may ask another agent for an appropriate rule, describing the situation if necessary. The rule obtained can be stored in the agent's knowledge base for a future use to eliminate communication needs in the future.

In this research, we make the following contribution: rule induction can be used for efficient strategy generation instead of reinforcement learning and evolutionary computation, rules produced are more readable, and learned knowledge can be easily exchanged between agents.

The following two application domains are used to test this idea: Fish Banks game, in which agents run fishing companies and have to decide where to fish, and Predator-Prey domain, in which agents are predators and learn how to catch a prey. This work is an extension of two conference papers about rule induction in Fish Banks game [22] in which supervised learning and reinforcement learning methods are applied to generate strategy in a this game, and [23] describing the results of rule induction application in Predator-Prey domain.

In the following section, related research is presented. Next, learning agent architecture is explained. The following section describes Fish Banks and Predator-Prey domains. Experimental results and their analysis conclude the work.

2 LEARNING IN MULTI-AGENT SYSTEMS

The most popular learning technique in multi-agent systems is reinforcement learning, which allows us to learn agent strategy: what action should be executed in a given situation. Other techniques can be also applied: neural networks, models coming from game theory as well as optimization techniques (like the evolutionary approach, tabu search, etc.). However, optimization techniques improve performance of the system using many populations of agents instead of a single agent experience, and should be considered as a separate class of algorithms.

A typical example of domain used in works on learning in multi-agent systems is Predator-Prey environment. In [27] the solution using reinforcement learning in this domain is presented. Predator agents use reinforcement learning to learn a strategy minimizing the time to catch a prey. Additionally, agents can cooperate by exchanging sensor data, strategies, or episodes. Experimental results show that cooperation is beneficial. Other researchers working on this domain successfully apply genetic programming [8] and evolutionary computation [6].

There is a large number of other works using reinforcement and evolutionary learning strategies. A good survey can be found in [17] and [19].

There is only a small number of works known to the author on supervised learning in multi-agent systems. They are presented below because they are closely related to the proposed learning mechanism.

Rule induction is used in multi-agent solutions for vehicle routing problems [5]. However, in this work learning is done off-line. First, rules are generated by the AQ algorithm (the same as used in this work) from traffic data. Next, agents use these rules to predict traffic.

Airiau et al. [1, 20] add learning capabilities to the BDI model. Decision tree learning is used to support plan applicability testing. Each plan has its own decision tree to test if it may be used in a given context. As a result, plans may be modified by providing additional conditions limiting its applicability. Knowledge learned has an indirect impact on the agent strategy because it has influence on probability of choosing plans for execution.

There are several works in wich Inductive Logic Programming (ILP) is applied. A good background paper considering machine learning and especially ILP for multi-agent systems is [9]. Rule induction (the subject of this paper) can be considered as a special case of ILP, where simple logic program defining one predicate only is learned. ILP shows its advantage over classical rule induction in complex domains, whereas most multi-agent applications are relatively simple (see conclusions of [9]). Therefore, rule induction seems to be enough in most cases.

It is also possible to combine several learning techniques in one agent. In [26] agents use reinforcement learning in a Vehicle Routing problem. Rule induction is used to decrease the size of the search space for reinforcement learning. If it is possible to learn a classifier for several attributes, these attributes can be replaced by a single value representing the class, which makes the space smaller.

Comparison of various learning strategies can be found in [24]. The paper presents a Farmer-Pest domain, which is especially designed for learning agent benchmarking. Every agent controls a farmer that supervises multiple fields. Several types of pests can appear on each field, and the farmer should execute an action which is appropriate for the pest type. Each pest is described by a set of attributes (e.g. number of legs, color) which are visible to the agent, while the pest type is hidden. Therefore, agents have to learn how to assign an action to the observed conditions represented by the attributes. In the paper agents using reinforcement learning (SARSA) are compared with those using supervised learning algorithms (Naïve Bayes, C4.5 and RIPPER).

3 LEARNING AGENT ARCHITECTURE

In this section we present learning agent architecture, which is general and fits various learning strategies. It is presented in Figure 1. Agent consists of four modules:

Processing Module which is responsible for basic agent activities, storing training data, executing learning process, and using learned knowledge.

Learning Module which is responsible for the execution of learning algorithms and giving answers for problems with the use of learned knowledge.

Training Data which is a storage for examples used for learning.

Generated Knowledge which is a storage for learned knowledge.

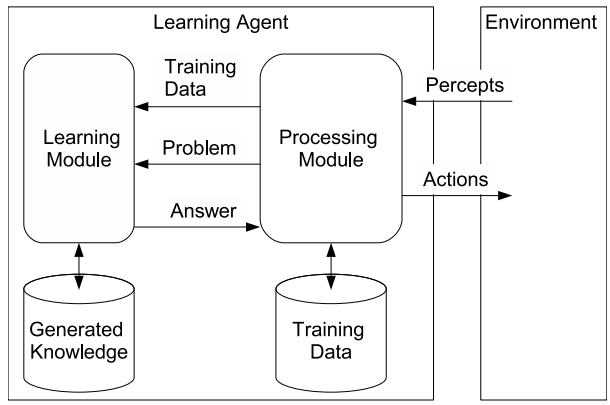

Figure 1. Learning agent architecture

These components interact in the following way. *Processing Module* receives *Percepts* from the environment. It may process them and execute *Actions*. If the learned knowledge is needed during processing, the module formulates a *Problem* and

sends it to the *Learning Module*, which generates an *Answer* for the *Problem* using *Generated Knowledge*. *Processing Module* also decides what data should be stored in the *Training Data* storage. When necessary (e.g. periodically, or when *Training Data* contains many new examples) it executes a learning algorithm in the *Learning Module* to generate new knowledge from *Training Data*. Learned knowledge is stored in the *Generated Knowledge* base.

Details of the components are domain-specific. In the context of this paper, the learning module is most interesting. It may be defined as a four-tuple: (*Learning Algorithm*, *Training Data*, *Problem*, *Answer*). Characteristics of the training data, the problem and the answer depend on the learning strategy used in the learning module. It makes the choice of the *Learning Algorithm* the most important.

Two types of learning modules have been developed and tested so far: the reinforcement learning module and the inductive rule learning module, although other learning methods can also be used. Below the learning modules for three types of popular learning strategies are characterized. Basic information about these learning strategies is reminded to show how they fit to the proposed architecture.

3.1 Reinforcement Learning

The most popular learning method in multi-agent systems is reinforcement learning. In this method, an agent gets a description of the current state and using its current strategy chooses an appropriate action from a defined set. The action is executed and a reward from the system is received. Next, the agent updates its strategy using a reward from the environment and the next state description. Several methods of choosing the action and updating the strategy have been developed so far. For example, in Q-learning developed by Chris Watkins [28] the action with the highest predicted value (Q) is chosen. Q is a function that estimates the value of the action in a given state:

$$Q : A \times X \to \mathbb{R}, \tag{1}$$

where A is a set of actions, X is a set of possible states, and $\mathbb{R}$ is a set of real numbers. Q function is updated after action execution:

$$Q(a,x) := Q(a,x) + \beta\Delta \tag{2}$$

$$\Delta = \gamma Q_{max} + r - Q(a,x) \tag{3}$$

$$Q_{max} = \max_a Q(a,x') \tag{4}$$

where $x, x' \in X$ are subsequent states, $a \in A$ is an action chosen, r is a reward obtained from the environment, $\gamma \in [0,1]$ is a discount rate (importance of the future rewards), and $\beta \in (0,1)$ is a learning rate. Various techniques are used to prevent getting into a local optimum. The idea is to explore the solution space better by choosing not optimal actions from time to time (e.g. random or not performed in a given state yet).

A reinforcement learning module can be responsible for managing all the agent activities or only a part of it (it can be activated in some type of states or can be responsible for selected actions only). The *Problem* definition is a description of the current state. The *Answer* is an action chosen using the current strategy (current Q function). *Training data* consists of the next state description (after executing action returned by the module), and of a reward. The reward may be observed by the agent or may be calculated by the processing module using some performance measures. *Generated Knowledge* is a Q function representation. It may be a simple table or a complex function approximator (e.g. a neural network).

3.2 Supervised Learning

Generally, supervised learning allows us to generate an approximation of a function $f : X \to C$ which assigns labels from the set C to objects from set X. To generate knowledge a supervised learning algorithm needs labeled examples which consist of pairs of f arguments and values. Let us assume that elements of X are described by a set of attributes $A = (a_1, a_2, \ldots, a_n)$, where $a_i : X \to D_i$. Therefore $x^A = (a_1(x), a_2(x), \ldots, a_n(x))$ is used instead of x. If the size of C is small, like in this research, the learning is called classification, C is set of classes, and h is called classifier.

The supervised learning module gets a *Training data*, which is a set $\{(x^A, f(x))\}$, and generates hypothesis h, which is stored in the *Generated Knowledge*. *Problem* is a x^A, and the *Answer* is $h(x^A)$.

There are several supervised learning methods. They use various hypothesis representations, and various methods of hypothesis construction. One of the most popular classification algorithms is C4.5, an inductive decision tree learning algorithm developed by Ross Quinlan [18]. C4.5 uses decision trees to represent h. The basic idea of learning is as follows. The tree is learned from examples recursively. If (almost) all examples in the training data belong to one class, the tree consisting of the leaf labeled by this class is returned. In the other case, the best attribute for the test in the root is chosen (using an entropy measure), training examples are divided according to the selected attribute values, and the procedure is called recursively for every attribute test result with the rest of the attributes and appropriate examples as parameters.

Another learning algorithm with a broad range of abilities, which was used in the implemented system is AQ. It was developed by Ryszard Michalski [15]. Its subsequent versions are still being developed. This algorithm also generates classifier from the training data, but h is represented by a set of rules which have tests on attribute values in the premise part, and a class in a conclusion. Rules are generated using sequential covering: the best rule (e.g. giving a good answer for the most examples) is constructed by a beam search, examples covered by this rule are eliminated from a training set, and the procedure repeats. In order to learn a rule for specific class AQ starts with one example of this class which is called a seed. It generates a star which is a set of maximally general rules covering the seed and

not covering any examples from other classes. This is done by repeating of the seed description generalization along successive attributes. Results of generalizations are intersected and the best rules are selected according to specified criteria. In this research off-the-shelf AQ21 program is used [29] and no modifications to the learning algorithm are made.

Other methods, using different knowledge representation, such as support vector machines, Bayesian or instance-based models also fit the above specification. Similarly, learning modules using artificial neural networks for classification or function approximation have the same input and output.

What is important, in the case of supervised learning, the processing module should provide a proper function value $f(x)$ for examples in the training data. If we are not able to provide this, inductive learning can not be used. However, if an agent has at least some qualitative information about $f(x)$ for given x^A (it can say if it is good or bad), it can use this information as a label and can build a classifier. Details of this work-around on a specific example can be found in Section 4.2.

3.3 Unsupervised Learning

In unsupervised learning the task of the learning module is to organize examples into groups called clusters, whose members are similar in a way. Examples of this strategy are Kohonen neural networks and clustering. *Training data* have a form of example descriptions $\{x^A\}$, without any label. The *Problem* is an example description x^A, and the *Answer* is the example's cluster identifier. *Generated Knowledge* stores neural network data or clusters definitions. This learning strategy is not commonly used in multi-agent systems.

4 FISH BANKS GAME

The Fish Banks game is originally designed for teaching people of effective cooperation in using natural resources [13]. It also may be applied as an environment for multi-agent simulations [10, 25].

The game is a dynamic environment providing resources, action execution procedures, and time flow represented by game rounds. Each round consists of the following steps: ships and money update, ship auctions, trading session, ship orders, ship allocation, fishing, and fish number update.

Agents represent players that manage fishing companies. Each company aims at collecting maximum assets expressed by the amount of money deposited at a bank account and the number of ships. The company earns money by fishing at fish banks. The environment provides two fishing areas: a coastal and a deep sea ones. Agents can also keep their ships at the port. The cost of deep sea fishing is the highest. The cost of staying at the port is the lowest but such ships do not catch fish. Initially, it is assumed that the number of fish in both banks is close to the bank's maximum

capacity. During the game the number of fish in every bank changes according to the following equation:

$$f_{t+1} = f_t + bf_t\left(1 - \frac{f_t}{f_{max}}\right) - C_t, \tag{5}$$

where f_t is a fish number at a time t, b is a birth rate (0.05 value was used in experiments), f_{max} is a maximum number of fish (equal to 4 000 for the deep sea, and 2 000 for the coastal area), $C_t = nc_t$ is a total fish catch: n is a number of ships of all players sent to the bank, and c_t is a fish catch for one ship at the time t:

$$c_t = c_{max} w_t \sqrt{\frac{f_t}{f_{max}}}, \tag{6}$$

where c_{max} is a maximum catch (equal to 25 for the deep sea, and 15 for the coastal area), and w_t is a weather factor at a time t, which is a random number between 0.8 and 1.0.

Initially, $f_1 = f_{max}$. Therefore, at the beginning of every game, f_t is close to f_{max}, and fishing in the deep sea is more profitable. During the game, exploration usually overcomes birth and after several rounds the number of fish can decrease to zero. It is a standard case of "the tragedy of commons" [7]. It is more reasonable to keep ships at the harbor then, therefore companies should change theirs strategies.

In the original game, fishing companies may order new ships to be built and may cross-sell their ships. The ships may also be sold at an auction organized by the game manager. In the current version of the system ship auctions and trading sessions are not supported.

All costs (of building a ship, its maintenance and use) and price of fish are constant for the whole game. At the end of the game the value of the ships owned by the companies is estimated and added to the money balance.

4.1 Agents

Four agent types are implemented in this domain: reinforcement learning agent, rule learning agent, predicting agent, and random agent. To allocate ships the first one uses a strategy generated by Q-learning, the second one uses rules induced from the experience, the third agent type uses previous fishing results to estimate values of possible allocation actions, and the last one allocates ships randomly.

All types of agents may observe the following aspects of the environment: arrival of new ships bought from a shipyard, money earned in the last round, all agents' ship allocation actions, and fishing results (c_t) for the deep sea and the inshore area. All types of agents can execute the following two types of actions: order ships, allocate ships.

Order ships action is currently very simple. It is implemented by all types of agents in the same way. At the beginning of the game every agent has 10 ships. In

every round, if there are less than 15 ships, there is a 50 % chance that two new ships will be ordered.

To allocate ships the agent has to decide how many of them are kept in the harbor and how many are sent to the fishing areas. Therefore, the allocation action is represented by a triple (h, d, c), where h is the number of ships left in a harbor, d and c are the number of ships sent to the deep sea, and to the coastal area, respectively. Ship allocation is based on the method used in [10]. Agents generate a list of allocation actions for $h = 0\,\%, 25\,\%, 50\,\%, 75\,\%$, and $100\,\%$ of the ships that belong to the agent. The rest of the ships (r) are partitioned; for every h the following action candidates are generated:

1. All: $(h, 0, r), (h, r, 0)$ – send all the remaining ships either to the deep sea or to the coastal area,
2. Check: $(h, 1, r-1), (h, r-1, 1)$ – send one ship to the deep sea or to the coastal area and the rest to the other,
3. Three random actions: $(h, x, r-x)$, where $1 \leq x < r$ is a random number – allocate remaining ships in a random way,
4. Equal: $(h, r/2, r/2)$ – send equal number of ships to both areas (one more ship is sent to a deep sea if r is odd).

The random agent allocates ships using one of the candidates chosen at random. The predicting agent uses the following formula to estimate the value of the action:

$$v(a) = income(a) + \eta ecology(a), \tag{7}$$

where $income(a)$ represents the prediction of the income under the assumption that in this round fishing results will be the same as in the previous round, $ecology(a)$ represents the ecological effects of the action a (the value is low if fishing is performed in the area with a low fish population, see [10] for details), and η represents the importance of the ecology factor. The action with the highest v value is chosen to execute. If there are several actions with the same, largest v value, the action generated earlier (with the lowest h value) is chosen.

4.2 Learning Agents

In this domain two learning strategies are used: reinforcement learning and rule induction. Although it is not our goal to compare these learning strategies (learning agents use different input for learning), results suggest that in some cases rule induction may improve performance faster than reinforcement learning. It needs further investigation, though.

For the reinforcement learning agent the *Problem* consists of the description of the current state $x \in X = \{(dc, cc) : dc \in \{1, 2, \ldots 25\}, cc \in \{1, 2, \ldots, 15\}\}$. It represents the catch in both areas in the previous round. The *Answer* is an action $a \in Act = \{(h, d, c) : h, d, c \in \{0\,\%, 25\,\%, 50\,\%, 75\,\%, 100\,\%\}, d + c = 1\}$. *Training*

```
begin
    if it is the first game then
        Store Q = 0 in Generated Knowledge
    end
    a_1 := random action;
    execute a_1;
    foreach round r = 2, 3, ... do
        x_r := current state;
        if random() > the probability of exploration then
            a_r := random action;
        end
        else
            a_r := best action in the current situation calculated using
            Generated Knowledge (arg max_a Q(a, x_r))
        end
        execute a_r;
        observe state after action execution x_{r+1};
        i:= income (money earned by fishing − costs);
        learn (update Q) using Training Data (x_r, a_r, x_{r+1}, i);
    end
end
```

Algorithm 1: Algorithm used by the reinforcement learning agent in the Fish Banks game

Data consists of the current and the next state description (after executing the action) and a reward. The reward is equal to the income of the agent after the fishing decreased by the ship maintenance costs. *Generated Knowledge* is a Q function tabular representation. Reinforcement learning agent chooses action at random in the first round. In the following rounds, the Q-learning strategy is used: the action with the highest predicted value (Q) is chosen. Details are presented in Algorithm 1.

At the beginning Q is initialized as a constant function 0. To provide sufficient exploration, in a game number g a random action is chosen with probability $1/g$ (all actions have the same probability). Therefore, the random or the best action (according to Q function) is chosen and executed.

For the agent using rule induction *Training Data* is a set $\{(x^A, f(x))\}$, where $x^A \in X \times Act$ is a state and ship allocation description, and $f(x) \in \{good, bad\}$ represents quality of the ship allocation in a given state. In every round the best ship allocation (among actions executed by all players) together with the current state description is classified as *good* and added to the *Training Data*. The worst ship allocation is classified as *bad* and is also added. Here availability of an action quality is important. It is worth noting that in such an approach no external teacher is necessary. It is an agent itself which labels examples.

```
begin
    if it is the first game then
        Generated Knowledge := ∅;
        Training Data := ∅
    end
    foreach round r do
        if it is the first game or round then
            a := random action
        end
        else
            a := action with the highest rating calculated using rules stored in
            Generated Knowledge
        end
        execute a;
        observe actions of other agents and results;
        add to Training Data examples {(best-action-data, good),
        (worst-action-data, bad)};
    end
    learn from Training Data;
    store knowledge in Generated Knowledge;
end
```

Algorithm 2: Algorithm used by the rule learning agent in the Fish Banks game

To find the best action in the current state, processing module iterates through all possible actions $a \in Act$, formulates *Problem* (x^A), and gets two numbers from the *Learning module*: $good(a)$ and $bad(a)$, which are numbers of rules stored in *Generated Knowledge* matching the action and current environment parameters, with consequence *good* and *bad*, respectively. Every action a gets a rating value v according to the formula:

$$v(a) = \alpha good(a) - bad(a), \tag{8}$$

where α is a weight representing the relative importance of rules with consequence *good*.

The rule learning agent not only randomizes actions in the first game, but in the following games it chooses actions with the highest rating. Behavior of the agent is presented in Algorithm 2.

If there is more than one action with the same value, the one occurring earlier in the list is chosen. As a consequence, actions with smaller h are preferred.

At the end of each game the learning agent uses *Training Data*, which contains examples generated during all games played so far, to learn a new classifier, which is used in the next game.

The software used in experiments was written in Prolog, using Prologix compiler [12]. Every agent is a separate process. It can be executed on a separate ma-

chine. Agents communicate with the environment using Linda blackboard. Prologix is an extension of BinProlog that has many powerful knowledge-based extensions (e.g. agent language LOT, Conceptual Graphs and KIF support). As mentioned, to support rule induction the AQ21 program is used. It is executed from Prolog and the rules generated are added to the code as additional Horn clauses.

5 PREDATOR-PREY DOMAIN

Predator-prey domain is a simulation with two types of agents: predators, and preys. The aim of a predator is to hunt for a prey. Environment is a grid world with size $n \times n$ with joined opposite edges (it is a torus). Time is discrete, and its flow is represented by turns. In every turn all agents receive percept data from the environment and chose their actions, which are executed next. The schema of the simulation environment is presented in Figure 2.

Percept data is provided for predator agents only. It consists of the closest prey's type and relative position. Predators have limited range of sight, and only preys which are closer than a given threshold are seen.

Agents can either move in the four directions (up, down, left, and right) or not move at all. If predator occupies a field next to the prey, the prey is captured.

To make the domain more complicated, two types of preys are defined: bird and mouse. The first one moves up and down with equal probability. Similarly, mouse moves left and right.

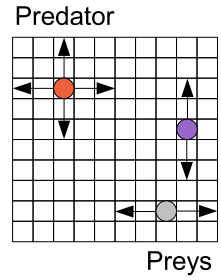

Figure 2. Predator-prey environment

Also, two types of predators are defined: random and learning. The former moves in four directions or does not move with equal probability. The latter uses rule induction to improve performance.

Game is defined as a sequence of turns beginning with the initial positions of agents and ending in a turn when all preys are captured.

```
begin
    if it is the first game then
        Generated Knowledge := ∅; Training Data := ∅
    end
    foreach round r do
        if it is the first game or round then
            a := random action
        end
        else
            a := action determined by the first matching rule from Generated
            Knowledge or action with the highest rating
        end
        execute a;
        observe the environment;
        if distance to the prey decreased then
            add example (current-situation, a) to Training Data
        end
    end
    learn from Training Data;
    store knowledge in Generated Knowledge;
end
```

Algorithm 3: Algorithm used in games by the rule learning predator

5.1 Learning Predator

During the simulation *Processing Module* receives *Percepts* and transforms them into a *Problem*. Currently the *Problem* has not more information than *Percepts*, but generally it may contain more data. *Learning Module* generates an *Answer* for the *Problem*, which is an action, which should be executed. To choose the action it uses *Generated Knowledge* stored in the knowledge base. This knowledge has a form of rules. The action is then executed. The behavior of the learning predator is presented in Algorithm 3.

Percepts received by the agent are equal to *null* if no prey is in the observation range, or is a triple (t, dx, dy), where t is a type of a prey, and dx, dy are relative coordinates of the prey along the X and Y axis, respectively.

If, after action execution, the distance to the prey decreases, the new example is stored in the *Training Data* memory. Attribute values are created using percept triple from the previous round and the action executed is used as a class. Here the important property of the environment is used. It is possible to generate training data because action results are visible immediately.

During a learning phase *Training Data* is sent to the *Learning Module*. In this domain the AQ rule induction algorithm is also used for learning.

If there are more examples in the *Training Data* with the same values of attributes and different classes (actions), learning algorithm assigns the majority class to the example.

Rules generated during learning and stored in the *Generated Knowledge* have the following form:

$$p_1, p_2, \ldots, p_n \rightarrow a, \tag{9}$$

where p_i are tests on the attributes representing the problem, and a is an action, which should be executed. In this domain learned rules can be applied in two ways:

1. First match – the first rule matching the *Problem* is chosen, and its consequence determines the action.
2. Counting – system selects the action, which is the most frequent in the consequences of rules matching the *Problem*. In the case of a draw, the winning action is chosen at random from the most frequent ones.

The agent executes a random action if there is no rule matching the problem, which is always the case at the beginning, when *Generated Knowledge* is empty, but it also happens later.

Because in this multi-agent system more than one entity generates the similar knowledge, impact of *communication* can be taken into account, and exchange of learned knowledge is tested. If there is no rule that matches a current state in the agent knowledge base, the agent can ask another agent for help. It sends the state description and receives matching rules if such exist. These rules can be used to generate an action for the current situation and are stored in the knowledge base for future use.

The software used in experiments is also written in Prolog, using Prologix compiler, and also AQ21 program is used for learning.

6 EXPERIMENTAL RESULTS

To test how learning influences the agent performance, experiments were performed in both domains. The hypothesis tested is that agents using rule induction achieve better performance while more experience is accumulated. Simulations have to be repeated many times because of the stochastic character of these domains. It is also necessary to check the statistical significance of the results.

6.1 Fish Banks Domain

Four experiments were performed in this domain. Four agents took part in every experiment. Each experiment consisted of the sequence of ten games and was repeated twenty times. The experience of the learning agents was collected during these sequences of games but, of course, it was cleared between repetitions. For every graph, the X axis represents the number of game in the sequence. Y axis

represents agent performance. Performance of an agent in a game is defined as the amount of money owned by the agent at the end of this game. Performance is measured in discrete points, after games. The measure points are connected with lines to improve readability.

In the first experiment there were three random agents and one reinforcement learning agent (with $\gamma = 1$ and $\beta = 0.1$). Performance of agents is presented in Figure 3.

Next, configuration with three random agents and one rule learning agent (with weight $\alpha = 1$) was tested. Average performance of agents in the consecutive games is presented in Figure 4. Learning with $\alpha = 2$ gives similar results.

In the third experiment there were two random agents, one reinforcement learning agent (with $\gamma = 1$ and $\beta = 0.1$) and one rule learning agent. Performance of agents is presented in Figure 5.

Finally, one learning ($\alpha = 1$), one predicting and two random agents were used. Performance of agents is presented in Figure 6.

In all of the experiments the average performance of both types of learning agents grows with the agent's experience, while the performance of the predicting and random agents decreases slightly (because of the learning agents competition). The reinforcement learning agent is a little bit worse than a rule learning agent, but tuning of its parameters (β, γ) and taking into account actions of other agents during learning should increase its performance.

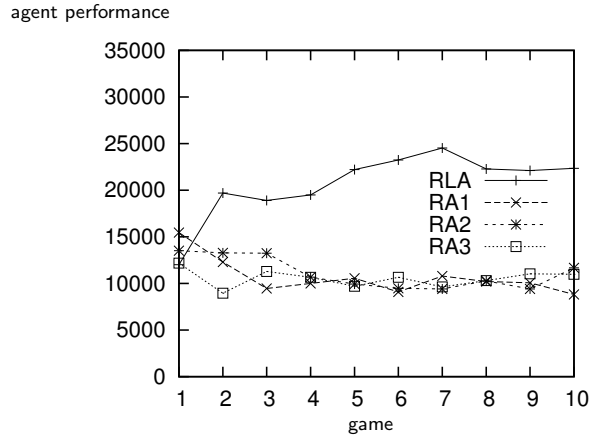

Figure 3. Comparison of performance of reinforcement learning agent (RLA) and agents using random strategy of ship allocation (RA1, RA2, RA3) in consecutive games (the higher, the better)

The standard deviation of the learning agents performance during experiments is relatively small. It is between 5 000 and 6 000 for random agents, about 5 000 or less for rule learning agents, and between 6 000 and 8 000 for reinforcement learning agents.

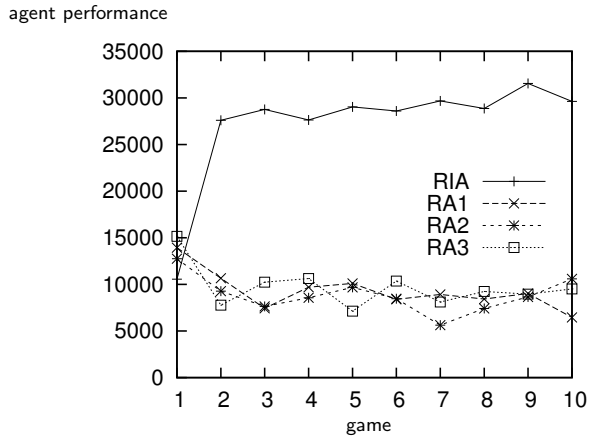

Figure 4. Comparison of performance of rule induction learning (RIA) and agents using random strategy of ship allocation (RA1, RA2, RA3) in consecutive games (the higher, the better)

The figures show clearly that learning improves the performance. However, statistical significance of performance differences was checked using the t-test. The first (random) result and also the result of games in which the rule learning agent has the best performance are selected for comparison. It is game No. 7 for the first experiment, No. 9 for the second and No. 8 for the remaining two experiments. Every difference is statistically significant at $p < 0.05$.

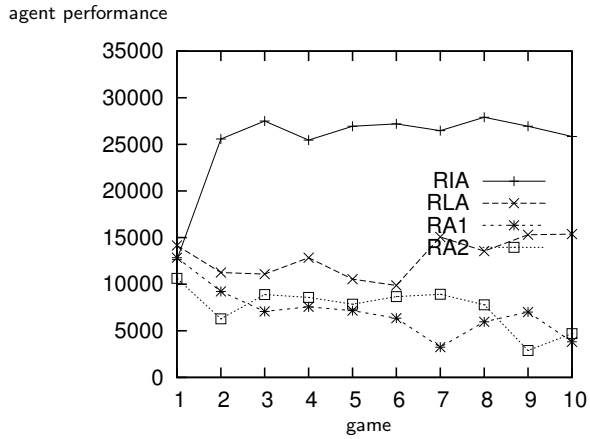

Figure 5. Comparison of performance of rule induction learning (RIA), reinforcement learning agent (RLA), and agents using random strategy of ship allocation (RA1, RA2) in consecutive games (the higher, the better)

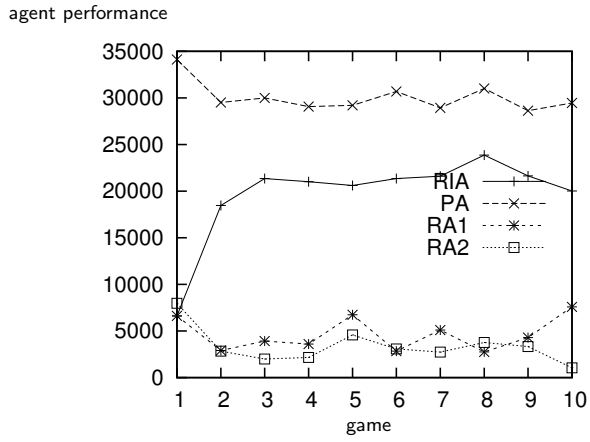

Figure 6. Comparison of performance of rule induction learning (RIA), prediction agent (PA), and agents using random strategy of ship allocation (RA1, RA2), the higher, the better

Experimental results show that the rule learning agent performance increases rapidly at the beginning of the learning process, when generated rules are used instead of a random choice. Next it increases slowly, because later examples do not bring significantly new information. The performance stabilizes at the end of the process.

As we can see in Figure 5, the rule learning agent performs better than the reinforcement learning agent, but the predicting agent outperforms the rule learning agent. Further research is necessary to determine if it is possible to learn such a good strategy.

Examples of rules learned are presented in Figure 7. They have the form of Prolog clauses. Capital letters represent variables that can be unified with any value. Predicate `member` checks if its first argument belongs to the list that is a second argument. It is used to represent an internal disjunction (expression of the form $x = v_1 \vee v_2 \vee \ldots \vee v_n$). These rules can be interpreted in the following way.

Clause (a): It is a bad decision to keep at the harbor 25, 50, or 75 percent of ships if the previous catch at the deep sea is greater than or equal to 16, and the previous catch at the coastal area is 10.

Clause (b): It is a good decision to send 100 % ships to the deep sea or 75 % to the deep sea and 25 % to the coastal area if the previous catch at the deep sea is greater than or equal to 18, and smaller than or equal to 21, and the previous catch at the coastal area is smaller than or equal to 10.

a)

```
rate(bad) :-
  harbor(B),
  member(B,[25,50,75]),
  prevCatchDeep(C),
  C >= 16,
  prevCatchCoastal(10).
```

b)

```
rate(good) :-
  alloc(B),
  member(B,[100%-0%,
    75%-25%]),
  prevCatchDeep(C),
  C >= 18,
  C =< 21,
  prevCatchCoastal(D),
  D =< 10.
```

Figure 7. Examples of rules (in the form of Prolog clauses) learned by the agent

6.2 Predator Prey Domain

Grid world in experiments had dimensions 24×24. Initial positions of predators were $(0, 0)$ and $(12, 12)$. There were two prey agents: bird and mouse. The first one started in position $(16, 16)$ with probability 0.8 and in $(4, 4)$ with probability 0.2, and the second one – vice versa. The reason for such a starting position was to simulate the situation, in which for every agent one of the states was more typical than the other. It made communication more profitable. Predator's range of sight was equal to 7.

The following three configurations were tested in experiments:

1. Learning predators using the first matching rule without communication;
2. Learning predators using counting of matching rules without communication;
3. Learning predators using the first matching rule with communication;

In all experiments 100 sequences of games were executed. Every sequence consisted of 16 consecutive games. *Training Data* and *Generated Knowledge* of learning predators were kept between games in a sequence. However, they were cleared between sequences. Performance of predators (represented in graphs by axis Y) was measured by a number of turns in a game needed to capture all preys; the less, the better. Maximum number of turns was 1500 even if some prey was not captured. Because of the implementation limitations, predators executed learning algorithm at the end of every even game. Therefore, performance measures are averaged in consecutive pairs of games, i.e. the first performance result is for games $(1, 2)$, the second one is for $(3, 4)$, and so on. In the graphs the X axis represents the number of the pair of games in the sequence.

In the first experiment predators using the first match rule application were tested. Average performance measures and its standard deviation in consecutive games are presented in Figure 8.

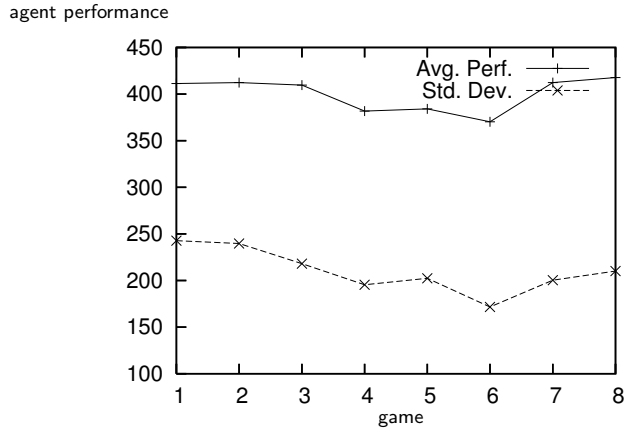

Figure 8. Average performance and its standard deviation for learning agents using first matching rule application in consecutive games (the lower, the better)

The second experiment tested predators using the counting rule application. Average performance measures and its standard deviation in consecutive games are presented in Figure 9.

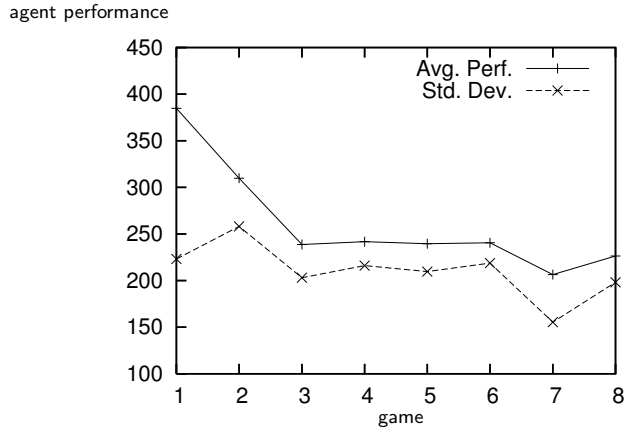

Figure 9. Average performance and its standard deviation for learning agents using counting rule application in consecutive games (the lower, the better)

The last experiment shows the influence of communication. Predators using first matching rule application with communication were tested. Average performance measures and its standard deviation in consecutive games are presented in Figure 10.

The average performance of random predators is equal to 403, and the standard deviation is 224. As we can see, agents using the first matching rule application

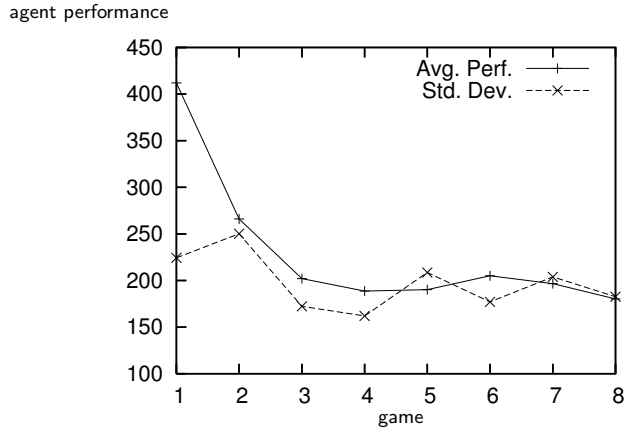

Figure 10. Average performance and its standard deviation for learning agents using first matching rule application with communication in consecutive games (the lower, the better)

improve the performance very slightly. The difference is statistically not significant because of the relatively high standard deviation. Agents using counting rule application and the first matching rule application with communication make much larger improvements. For these two experiments statistical significance of performance differences between the first step and the best one was checked using the t-test. The best results are achieved in step 7 for counting rule application and in step 8 for the first matching with communication. Both differences are significant at $p < 0.05$.

Like in the previous domain, performance improves rapidly at the beginning and after several rounds it stabilizes, because of the lack of new knowledge contained in the following examples.

Examples of rules learned by the predator-agent in a form of Prolog clauses are presented in Figure 11. Both rules have action `left` in the conclusion. `S` is an identifier of a state for which decision should be made. Rule a) is applicable if the predator sees a bird, which has the same or smaller by one x coordinate and relative vertical position is between -3 and 0. The premise of rule b) determines if the predator sees a mouse, whose relative x coordinate is -2, and y is between -3 and -2.

7 CONCLUSION AND FURTHER RESEARCH

In this paper the idea of using rule induction for generating strategy for agents is presented. It is tested on two domains: Fish Banks and Predator-Prey. However, inductive learning can be applied in any domain, in which a learning agent is able to observe the results of its actions (there is no reward assignment problem).

a)

```
dir(S,left) :-
  type(S,bird),
  dx(S,X),
  X >= -1,
  X =< 0,
  dy(S,Y),
  Y >= -3,
  Y =< 0.
```

b)

```
dir(S,left) :-
  type(S,mouse),
  dx(S,-2),
  dy(S,Y),
  Y >= -3,
  Y =< -2.
```

Figure 11. Examples of rules (in the form of Prolog clauses) learned by the agent

Not only rule induction can be applied here, other supervised learning methods, like decision trees or Bayesian networks, will work too, although, because rule-based knowledge representation is modular, the exchange of the knowledge between agents is easy and has low cost. Only necessary rules can be transmitted.

Another advantage of rule induction is clarity of rule-based knowledge representation. It is possible to interpret the knowledge base and check its correctness manually. It is very important for some domains.

In the future this method of strategy generation should be applied in other domains, like crisis management [3], grid monitoring [2], QoS provisioning for data intensive applications [21] or better development of knowledge-based grid organizational memory [11]. It is also important to test applicability in cases with delayed information about action impact to the goals. Rule induction should be also more broadly compared to other adaptation strategies, including evolutionary methods and others (see e.g. [4]).

Other methods of resolving ambiguity in the training data should be tested as well. Here the majority strategy is used, but other possibilities (including in positives or negatives and ignoring during learning) should also be examined. Exchange of training data as another form of communication during learning should be tested too.

More research should be also performed in mixing several learning methods in one agent. For different aspects of agent activity, specific learning methods could be used.

Acknowledgments

The research leading to the results described in the paper has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement No. 218086. The author is grateful to Arun Majumdar, Vivomind Intelligence Inc. for providing the Prologix system (used for implementation), and for help with using it, to Janusz Wojtusiak, MLI Laboratory, George Mason

University, for AQ21 software and assistance, to Jens Pfau, Technische Universität, Darmstadt for suggesting Predator-Prey domain, and implementation of the first, Java version of the system using decision-tree induction and reinforcement learning, and last but not least to Prof. R. S. Michalski for introduction into the machine learning domain. Statistical analysis was performed using Statistica package provided by Statsoft Poland.

REFERENCES

[1] Airiau, S.—Padham, L.—Sardina, S.—Sen, S.: Incorporating Learning in BDI Agents. In Proceedings of the ALAMAS+ALAg Workshop, May 2008.

[2] Balis, B.—Kowalewski, B.—Bubak, M.: Real-Time Grid Monitoring Based on Complex Event Processing. Future Generation Computer Systems, Vol. 27, 2011, No. 8, pp. 1103–1112.

[3] Byrski, A.—Kisiel-Dorohinicki, M.—Carvalho, M.: A Crisis Management Approach to Mission Survivability in Computational Multi-Agent Systems. Computer Science, Vol. 11, 2010, pp. 99–113.

[4] Cetnarowicz, K.—Drezewski, R.: Maintaining Functional Integrity in Multi-Agent Systems for Resource Allocation. Computing and Informatics, Vol. 29, 2010, No. 6, pp. 947–973.

[5] Gehrke, J. D.—Wojtusiak:, J.: Traffic Prediction for Agent Route Planning. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot (Eds.): ICCS (3), Volume 5103 of Lecture Notes in Computer Science, Springer 2008, pp. 692–701.

[6] Lee Giles, C.—Jim, K.-C.: Learning Communication for Multi-Agent Systems. In WRAC 2002, pp. 377–392.

[7] Hardin, G.: The Tragedy of Commons. Science, Vol. 162, 1968, pp. 12431248.

[8] Haynes, T.—Sen, I.: Evolving Behavioral Strategies in Predators and Prey. In Adaptation and Learning in Multiagent Systems, Springer Verlag 1996, pp. 113–126.

[9] Kazakov, D.—Kudenko, D.: Machine Learning and Inductive Logic Programming for Multi-Agent Systems. In Multi-Agent Systems and Applications, Springer, 2001. pp. 246–270.

[10] Kozlak, J.—Demazeau, Y.—Bousquet, F.: Multi-Agent System to Model the Fishbanks Game Process. In The First International Workshop of Central and Eastern Europe on Multi-agent Systems (CEEMAS '99), St. Petersburg 1999.

[11] Kryza, B.—Slota, R.—Majewska, M.—Pieczykolan, J.—Kitowski, J.: Grid Organizational Memory Provision of a High-Level Grid Abstraction Layer Supported by Ontology Alignment. Future Generation Computer Systems, Vol. 23, 2007, No. 3, pp. 348–358.

[12] Majumdar, A.—Tarau, P.—Sowa, J. F.: Prologix: Users Guide. Technical report, VivoMind LLC, 2004.

[13] Meadows, D.—Iddman, T.—Shannon, D.: Fish Banks, LTD: Game Administrator's Manual. Laboratory of Interactive Learning, University of New Hampshire, Durham, USA 1993.

[14] Michalski, R. S.: Attributional calculus: A Logic and Representation Language for Natural Induction. Technical report, George Mason University 2004.

[15] Michalski, R. S.—Larson, J.: Aqval/1 (aq7) User's Guide and Program Description. Technical Report 731, Department of Computer Science, University of Illinois, Urbana, June 1975.

[16] Newell, A.—Simon, H. A.: Human Problem Solving. Prentice-Hall 1972.

[17] Panait, L.—Luke, S.: Cooperative Multi-Agent Learning: The State of the Art. Autonomous Agents and Multi-Agent Systems, Vol. 11, 2005.

[18] Quinlan, J. R.: C4.5: Programs for Machine Learning. Morgan Kaufmann 1993.

[19] Sen, S.—Weiss, G.: Learning in Multiagent Systems. MIT Press, Cambridge, MA, USA 1999, pp. 259–298.

[20] Singh, D.—Sardina, S.—Padgham, L.—Airiau, S.: Learning Context Conditions for BDI Plan Selection. In Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10), Volume 1, pp. 325–332, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[21] Slota, R.—Krol, D.—Skalkowski, K.—Orzechowski, M.—Nikolow, D.—Kryza, B.—Kitowski, J.: Computer Science, Vol. 13, 2012, No. 1, pp. 63–73.

[22] Śnieżyński, B.: Resource Management in a Multi-Agent System by Means of Reinforcement Learning and Supervised Rule Learning. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot (Eds.): Computational Science – ICCS 2007, Lecture Notes in Computer Science, Springer, 2007, pp. 864–871.

[23] Śnieżyński, B.: Agent Strategy Generation by Rule Induction in Predator-Prey Problem. In ICCS 2009: Proceedings of the 9th International Conference on Computational Science, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg 2009, pp. 895–903.

[24] Śnieżyński, B.—Dajda, J.: Comparison of Strategy Learning Methods in Farmer Pest Problem for Various Complexity Environments without Delays. Accepted to Journal of Computational Science, 2011.

[25] Śnieżyński, B.—Kozlak, J.: Learning in a Multi-Agent Approach to a Fish Bank Game. In M. Pechoucek, P. Petta, and L. Z. Varga (Eds.): Multi-Agent Systems and Applications IV: 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2005), Budapest, Hungary, Volume 3690 of Lecture Notes in Computer Science 2005, pp. 568–571.

[26] Śnieżyński, B.—Wojcik, W.—Gehrke, J. D.—Wojtusiak, J.: Combining Rule Induction and Reinforcement Learning: An Agent-Based Vehicle Routing. In Proc. of the ICMLA 2010, Washington D. C., pp. 851–856.

[27] Tan, M.: Multi-Agent Reinforcement Learning: Independent vs. cooperative agents. In Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann 1993, pp. 330–337.

[28] Watkins, C. J. C. H.: Learning from Delayed Rewards. Ph. D. thesis, King's College, Cambridge 1989.

[29] Wojtusiak, J.: AQ21 User's Guide. Technical report, George Mason University, Fairfax, VA 2004.

Bartłomiej Śnieżyński received his Ph. D. degree in computer science in 2004 from AGH University of Science and Technology in Krakow, Poland. In 2004 he worked as a Postdoctoral Fellow under the supervision of Professor R. S. Michalski at the Machine Learning and Inference Laboratory, George Mason University, Fairfax, VA, USA. Currently, he is an Assistant Professor in the Department of Computer Science at AGH. His research interests include machine learning, multi-agent systems, and knowledge engineering.