Int. J. Appl. Math. Comput. Sci., 2015, Vol. 25, No. 3, 471–482
DOI: 10.1515/amcs-2015-0035

amcs

A STRATEGY LEARNING MODEL FOR AUTONOMOUS AGENTS BASED ON CLASSIFICATION

BARTŁOMIEJ ŚNIEŻYŃSKI [a]

[a] Department of Computer Science
AGH University of Science and Technology, al. Mickiewicza 30, 30-057 Kraków, Poland
e-mail: Bartlomiej.Sniezynski@agh.edu.pl

In this paper we propose a strategy learning model for autonomous agents based on classification. In the literature, the most commonly used learning method in agent-based systems is reinforcement learning. In our opinion, classification can be considered a good alternative. This type of supervised learning can be used to generate a classifier that allows the agent to choose an appropriate action for execution. Experimental results show that this model can be successfully applied for strategy generation even if rewards are delayed. We compare the efficiency of the proposed model and reinforcement learning using the farmer–pest domain and configurations of various complexity. In complex environments, supervised learning can improve the performance of agents much faster that reinforcement learning. If an appropriate knowledge representation is used, the learned knowledge may be analyzed by humans, which allows tracking the learning process.

Keywords: autonomous agents, strategy learning, supervised learning, classification, reinforcement learning.

1. Introduction

The most common learning technique used in agent-based systems to learn strategies is reinforcement learning (Panait and Luke, 2005; Sen and Weiss, 1999; Tuyls and Weiss, 2012). Only few works consider application of classification or, more generally, supervised learning. Reinforcement learning is a method that allows the agent to improve its strategy using feedback from the environment after the agent's action execution. The feedback is a scalar value, which represents a quality of the action. Algorithms of this type are simple, but the process of learning is relatively slow and without complicated extensions it is difficult to learn in environments having large state space (Kaelbling *et al*., 1996). This is a result of the *curse of dimensionality*, a well known problem of dynamic programming (Bellman, 1957), on which the reinforcement learning is based. Another disadvantage of this approach is the difficulty of analyzing the generated knowledge by humans. The second method of learning often used in agent-based systems or evolutionary computations relies on the processing of many agent generations, improving their performance in subsequent iterations (Panait and Luke, 2005). The necessity of maintaining many populations of agents makes it difficult to apply directly by a single, autonomous agent to learn its strategy online. Therefore, this type of learning is not considered here.

Given these observations, the main assertion of this paper is that *supervised classification can be successfully applied online by autonomous agents to learn their strategy effectively, using their own experience as a source of training data*.

The proposed model is based on artificial intelligence methods. It goes along the AI approach proposed by Shoham *et al*. (2003, pp. 7–8).

Classification is a type of supervised learning which generates a classifier from training data (a set of labeled examples). The training data are more than a feedback needed by reinforcement learning. However, in many domains the agent is able to create examples from its observations (Śnieżyński, 2013a). Supervised learning algorithms are more computationally demanding than reinforcement learning, but in the proposed solution the learning algorithm may be executed from time to time only (e.g., when the agent is idle), and in the tested domains the size of the training data was small (at most hundreds of examples).

In our research, we make the following contributions

to the state of the art: we propose a strategy learning model for autonomous agents based on classification; we show in experiments that this model can be applied for strategy generation; we compare the efficiency of supervised learning and reinforcement learning in configurations of various complexity and show that, in a complex environment, supervised learning can improve the performance of agents much faster that reinforcement learning.

In the first part of the paper an overview of the most important research results in the area of agent-based systems in the context of machine learning is provided. Against this background, the model allowing autonomous agents to generate their strategy with the use of classification is presented.

To cover cases with delayed results, the basic model is extended using the time window approach. An algorithm is defined to choose the action for execution.

Having the basic and extended models introduced, experimental results are presented. The effectiveness of learning agents using the proposed model and reinforcement learning are compared through the farmer–pest problem (Śnieżyński and Dajda, 2013) in several configurations characterized by growing complexity, for both immediate and delayed cases. The results show that the model proposed allows more rapid improvement of the agent's efficiency than the reinforcement learning approach. What is more, the learned strategy is described by the knowledge, which has a readable form.

The paper ends with concluding remarks summarizing the results and providing an outline of the most important contributions. Finally, directions for future research are given.

2. Related research

A good survey of learning in multi-agent systems working in various domains can be found in the works of Panait and Luke (2005), Sen and Weiss (1999) as well as Tuyls and Weiss (2012). The most popular learning technique applied is reinforcement learning (Sutton and Barto, 1998). The learning agent model assumes that the agent interacts with the environment in discrete steps, sets its state s by observing the environment, executes actions and receives a reward $r \in \mathbb{R}$. The reward is high if actions are good, and low if they are bad. The agent has to learn which action should be executed in a given state. The formal model of learning is based on a Markov process.

A lot of algorithms have been developed for this model. In the experiments we use the SARSA algorithm (Rummery and Niranjan, 1994). The strategy is represented by a function Q that estimates the quality value of the action in a given state: $Q : Act \times S \to \mathbb{R}$, where Act is a set of actions and S is a set of possible states. Knowing the current state $s_t \in S$ and using its current strategy, the agent chooses an appropriate action $a_t \in Act$. Usually, action with the highest Q value is chosen. Next, using reward r_t obtained from the environment, the next state description $s_{t+1} \in S$, and the action $a_{t+1} \in Act$ that will be next executed, it updates the Q function:

$$\Delta := r_t + \gamma Q(a_{t+1}, s_{t+1}) - Q(a_t, s_t), \quad (1)$$

$$Q(a_t, s_t) := Q(a_t, s_t) + \beta\Delta, \quad (2)$$

where $\gamma \in [0, 1]$ is a discount rate (importance of the future rewards) and $\beta \in [0, 1]$ is a learning rate. The reward characteristics depend on the problem. It represents the quality of the action. It has a high value in the case of achieving a goal, and low in the case of a failure.

To speed up the learning process, various techniques are developed. One of them is the temporary differences mechanism TD($\lambda > 0$) (Watkins, 1989), which updates not only the last state but also those visited recently. The parameter $\lambda \in [0, 1]$ is a recency factor. The values close to zero mean that traces are very short.

In reinforcement learning, various techniques are used to prevent computations from getting stuck into a local optimum. The idea is to explore the solution space better by choosing nonoptimal actions from time to time (e.g., random or not performed in a given state yet). In Boltzmann selection (Sutton and Barto, 1998) instead of selecting the action with the highest value, the action a^* is selected in state s with probability $P(a^*, s)$ calculated according to the following formula:

$$P(a^*, s) = \frac{e^{Q(a^*,s)/\tau}}{\sum_a e^{Q(a,s)/\tau}}, \quad (3)$$

where $\tau > 0$ is a temperature parameter. High values of τ make the probability of all actions almost the same, regardless their quality.

Supervised learning allows generating an approximation of some function $f : X \to Cat$ which assigns labels from the set Cat to objects from set X. To generate knowledge, a supervised learning algorithm needs labeled examples which consist of pairs of f arguments and values. If the size of Cat is small, like in this research, the learning is called classification, Cat is the set of classes (categories), and the learned approximation of f is called the classifier.

In the experiments we use three supervised learning algorithms: naïve Bayes (NB), C4.5 and RIPPER. NB is a simple probabilistic classifier, in which every attribute describing examples depends on the category. Learning is a process of calculation of *a priori* and conditional probabilities. C4.5 is a decision tree learning algorithm developed by Quinlan (1993). 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. Otherwise, the best attribute for the test in the root is chosen (using the 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 attributes and appropriate examples as parameters). RIPPER, developed by Cohen (1995), generates decision rules instead of trees. The classifier is represented by an ordered list of rules which have conjunction of tests on attribute values in the premise part and a class in a conclusion. During the growing phase, rule preconditions are extended by adding a new test to premises to give right decisions on training examples. After growing, the pruning phase is executed to avoid overfitting. In this phase, tests are removed from preconditions.

Reinforcement learning has been applied in many domains. One of them is the predator–prey domain. It is a simple simulation with two types of agents: predators and preys. The aim of a predator is to hunt a prey. The prey is captured if the predator (or several predators if cooperation is tested) is close enough. In the work of Tan (1993), predator agents use reinforcement learning to learn a strategy minimizing time to catch a prey. Additionally, agents can cooperate by exchanging sensor data, strategies or episodes.

Another environment is a grid world (Sutton, 1990), in which an agent can move vertically or horizontally and has to arrive to the goal state walking around obstacles. Extensions to this environment are added to make it more complex. For example, in the work of Lin (1992), energy sources and enemies are added. Also the taxi domain (Dietterich, 2000) can be considered an extension in which an agent has to transport passengers between selected locations.

In complex domains, reinforcement learning suffers from a computational problem caused by the large state space. Therefore, various techniques are proposed, like state generalization with the use of approximators, e.g., neural networks (Zhang and Dietterich, 1995), hierarchical decomposition (Dietterich, 2000), or Q-decomposition (Russell and Zimdars, 2003). Application of these methods makes the model much more complex. Another solution is proposed by Bazzan *et al*. (2011). Low-level agents are supervised by tutors that observe low-level agents. Supervisors store tuples representing joint states, actions and average rewards. Using this basis, they recommend actions to low-level agents. This allows agents to learn how to cooperate.

Many agent models and architectures have been developed so far (Wooldridge, 2009). An example of classification of models can be found in the work of Russell and Norvig (2009), where the following types are distinguished: a simple reflex agent, a model-based reflex agent, a goal-based agent, a utility-based agent and a general learning-agent. Only the last one is adaptable and similar to the solution proposed in this paper. The model proposed below is more specific because of the assumed supervised learning. A common architecture is BDI (belief-desire-intention) (Rao and Georgeff, 1991), in which beliefs represent the agent's current knowledge, desires represent its current objectives, and intentions represent the current goals that are chosen to be achieved by the agent.

There are few works on supervised learning applications in multi-agent systems. Some of them are discussed below. Rule induction is used in multi-agent solutions for vehicle routing problems (Gehrke and Wojtusiak, 2008). 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 global traffic data. Next, agents use these rules to predict traffic. There is an extension of this work (Śnieżyński *et al*., 2010). Agents use a hybrid learning algorithm. Rule induction is used to decrease the size of the search space for reinforcement learning.

Airiau *et al.* (2008) add learning capabilities to the above-mentioned BDI model (Singh *et al*., 2010). 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 their applicability. The knowledge learned has an indirect impact on the agent strategy because it has influence on the probability of choosing plans for execution.

In the work of Barrett *et al*. (2012) the C4.5 algorithm is used by the agent to build a model of teammates. Here also the predator–prey environment is used in experiments. A similar problem is discussed by (Hernandez-Leal *et al*., 2013), with the agent building models of other agents that change its strategy. Here also the C4.5 algorithm is used, but it is combined with an MDP (Markov decision process).

There are several works in which inductive logic programming (ILP) is applied. There is a good background paper considering machine learning and especially ILP for multi-agent systems (Kazakov and Kudenko, 2001). Supervised learning (the subject of this paper) can be considered a special case of ILP, where a 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 (conclusions of Kazakov's work). Therefore, supervised learning seems to be enough in most cases.

This work is based on several previous works (e.g., Śnieżyński, 2013a; 2013b; Śnieżyński and Dajda, 2013), in which supervised learning was applied to generate the

agent's strategy. However, in these papers, no formal model was defined. There were also no experiments with delayed action results and the entropy measure.

3. Model

A general idea of the model is presented in Fig. 1. The learning agent observes the environment and the observation is represented by percepts. The processing module has three main tasks: it updates the agent's current state, stores experience in the training data base and chooses an action to be executed by applying a classifier to an example representing the current percepts and the state. The classifier represents the agent's knowledge about the efficient strategy. It is generated by a supervised learning algorithm from the training data representing the agent's experience.

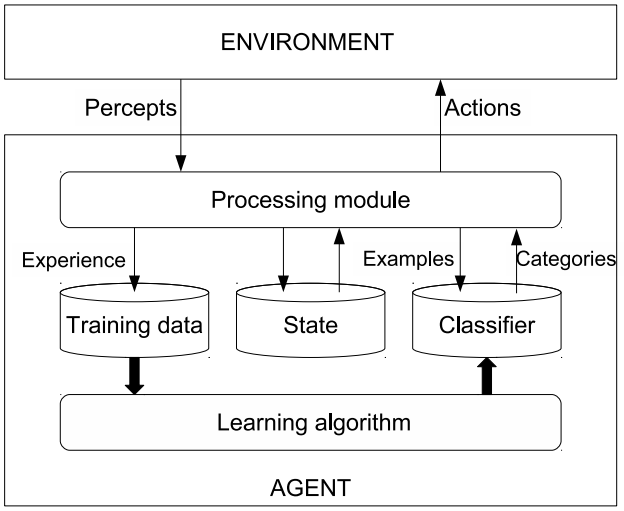


Fig. 1. Idea of the strategy learning model for autonomous agents.

3.1. Formal model. Let Per and Act represent sets of possible percepts and actions, respectively. S represents the set of the agent's states. The state may contain some information about past events.

To define how the experience is expressed, we use the following notation. The learning domain is represented by examples $x \in X$. Examples are described by attributes $A = (a_1, a_2, \ldots, a_n)$, where $a_i : X \to D_i$, and D_i is a domain of attribute a_i. Therefore, attribute values $x^A = (a_1(x), a_2(x), \ldots, a_n(x)) \in X^A$ are used instead of x. Every example x may be assigned to one of the categories: $cat(x) \in Cat$. Because we consider classification, we assume that $|Cat|$ is small. The experience is defined as a set of all possible labeled examples:

$$Exp = \{(x^A, cat(x))\}. \tag{4}$$

The experience is used as input data for a learning algorithm $L : 2^{Exp} \to C$, which generates a classifier $c \in C$. The classifier is used to assign the category to a given example: $c : X^A \to Cat$.

The agent's strategy learning model based on classification (SLMc) may be defined as the following tuple:

$$SLMc = (Per, Act, A, Exp, P, L, T_t, c_t, s_t, s_0, per_t), \tag{5}$$

where the first four symbols are defined above, P is the processing module, L is the learning algorithm, $T_t \subseteq Exp$ is the current training data (in time stamp t), $c_t \in C$ is the current classifier, s_t, s_0 are current and initial states, respectively, and $per_t \in Per$ are current percepts.

Taking into account the main tasks, the processing module P may be decomposed into the following three components:

$$P = (SU, EG, SM). \tag{6}$$

$SU : Per, S, Act \to S$ is a state update function used to determine the next internal state of the agent, taking into account percepts, the previous state and the last executed action.

$EG : Per, S \to Exp$ is an experience generator, which transforms percepts and the current state into a labeled example, which may be stored in the training data. It may be decomposed into two functions: description $D : Per, S \to X^A$ describing percepts and the state by attributes, and category determination $CD : Per, S \to Cat$, which chooses a category for the example, usually taking into account the observed results of the action chosen in the previous step or in the past.

$SM : Per, S \to Act$ is a strategy module, which uses the classifier c_t to choose the best action. Two approaches are considered in the model:

- Decision class (DC): category represents an action to be executed. Such a solution can be applied, when the number of actions to choose from is small. To choose the action for execution, SM applies description D to per_t and s_t, and uses the classifier c_t to determine the action for the example $x_t^A = D(per_t, s_t)$.

- Quality class (QC): the category represents the quality of action. In experiments, two categories are used: $Cat = \{good, bad\}$. If rewards are continuous, success may be defined as a reward above a certain threshold. In case of the QC, attributes A are used to describe percepts, the current state and the action. The QC approach fits environments in which the number of actions is high (see, e.g., Śnieżyński and Kozlak, 2005) or there are delays in observable action results (see the next subsection). To choose the action for execution, SM prepares examples

$x^A_{a_i} = D(per_t, s_{a_i})$ for every analysed action $a_i \in Act$: the previous action stored in s_t is replaced by a_i. Classifier c_t is used to classify all $x^A_{a_i}$ and SM chooses the action which has the best quality (the highest certainty of the *good* category). A classifier that is able to return the certainty of categories for a given example is needed in QC.

The SM definition should also contain some exploration strategy.

An integral part of the model is the algorithm of the agent, which is presented as Algorithm 1. At the beginning, the values of the current state (s_t), training data (T_t) and classifier (c_t) are initialized. Variable act representing an action from the previous round is set to the $null$ value. In the loop current observations variable is set (per_t). Next, state update function (SU) is used to calculate the current state value (s_t). An example is added to T_t (Lines 7–9) if action results are interesting to the agent, which depends on the application. In some domains (like in the one described below), all examples are stored. In other domains, some examples may be omitted, like in Fish–Banks (Śnieżyński and Kozlak, 2005), where only these actions that give very high or very low income are considered interesting. If the classifier is empty (Line 11), the action is random. Else, the action act is chosen using the strategy module (SM) and it is executed. Finally, from time to time (or when the agent is idle) the current classifier is updated (Line 17) using the learning algorithm (L) and training data (T_t).

The proposed model makes the strategy learning efficient even in environments with a large state space, because the computational complexity of supervised learning grows slower with the number of attributes than reinforcement learning, where adding one dimension to the state space causes exponential growth. What may be also important in some domains, if a symbolic knowledge representation is applied in the classifier (e.g., the decision tree or rules), it is also possible to analyze the generated strategy by humans.

3.2. Delayed rewards. Application of SLMc in environments in which results of action executions are observable immediately is straightforward. To cover cases with delayed results, an appropriate state representation using the time window should be applied. Also, a specific algorithm in SM should be used to choose the action for execution. An important part of the algorithm is the estimation of the impact of action variation (for a given range of time distances between action execution and the observed effects) on certainties of categories. The impact may be measured using entropy or maximal dispersion.

When a time window of size dt is used, the sequence of percepts and actions executed (of the length dt) are stored in the state. Therefore, every example x in time

Algorithm 1. Algorithm of the learning agent.

Require: Agent model $MSLMc = (Per, Act, A, Exp, P, L, T_t, c_t, s_t, s_0, per_t)$, where $P = (SU, EG, SM)$

1: $s_t := s_0$
2: $T_t := \varnothing$
3: $c_t := null$; $act := null$
4: **while** agent is alive **do**
5: $per_t :=$ observations of the environment
6: $s_t := SU(per_t, s_t, act)$
7: **if** results of the previous action are interesting **then**
8: $ex := EG(per_t, s_t)$
9: $T_t := T_t \cup \{ex\}$
10: **end if**
11: **if** $c_t = null$ **then**
12: act:= random action
13: **else**
14: $act := SM(per_t, s_t)$
15: execute act
16: **if** it is learning time (e.g., every 100 steps) **then**
17: $c_t := L(T_t)$
18: **end if**
19: **end if**
20: **end while**

stamp t represents a sequence and we use the following notation:

$$X \ni \overrightarrow{x_t} = (x_t, x_{t-1}, \ldots, x_{t-dt+1}). \tag{7}$$

Every x_i in the sequence is described by attributes A:

$$x^A_i = (a_1(x_i), a_2(x_i), \ldots, a_n(x_i), a(x_i)). \tag{8}$$

For simplicity, we assume that the last attribute $a(x_i) \in Act$ represents an action executed in time step i. However, several attributes may be also used to describe the action. Categories $Cat = \{good, bad\}$ are used to represent a success or failure of executing actions $a(x_i)$ in the sequence $\overrightarrow{x_t}$.

Because the QC approach is applied, the agent needs a classifier $c \in C$, which for a given sequence $\overrightarrow{x_t}$ returns the certainty of a given category y: $c(\overrightarrow{x}^A_t, y) \in [0, 1]$. The classifier should be also able to work with unknown attribute values.

SM has to select the best action $act^* \in Act$ for the current time t using c_t (Algorithm 1, Line 14). We propose to apply a special algorithm for SM (Algorithm 2). It works as follows:

1. Prepare $\overrightarrow{x_t}$ by setting all attributes describing the current and previous states and leaving action attributes unknown. If $t < dt$, then attributes describing non-existing states are also set to unknown value (Lines 1–5).

2. Consider consecutive possible delays. Variable $\overrightarrow{x}^{act}_{delay}$ is equal to $\overrightarrow{x_t}$ with state and percepts descriptions moved back $delay$ steps and filling newer time steps with an unknown values. Calculate δ_{delay} representing the entropy (or dispersion) of certainties of positive class for various actions executed with assumed delay (Lines 6–16).

3. Find delay del^* for which δ_{delay} suggests the highest impact of the action change (Lines 17–21).

4. Check the certainty of the category $good$ for various actions substituted at assumed del^* delay and return the one with the highest certainty (Lines 22–23).

4. Testing environment

We tested the proposed model on a farmer–pest problem[1]. This environment borrows the concept from a specific aspect of the real world, in which farmers struggle to protect their fields and crops from pests. Each farmer (this is the only type of agent in the problem) can manage multiple fields. On each field, multiple kinds of pests can appear. Each pest has a type assigned and a specific set of attributes e.g., the number of legs or the color. The values of these attributes depend on the pest type. To protect the field, the farmer can take advantage of multiple means (e.g., pesticides) called *actions*. However, each pest type has different resistance to each farmer's actions (hereinafter referred to as the *resistance matrix*). Usually, the problem is time-limited to a discrete number of turns. In every turn an agent can execute one action only. This makes simple strategies like applying all actions for every pest inefficient.

The key assumption here is that the farmer agent is not aware of the possible types of pests nor the resistance matrix. What he/she can see are the pests' attributes. Based on them, he/she needs to learn how to recognize different pest types. To learn the resistance matrix, the agent needs to experiment with different actions and observe their effects (i.e., whether or not the pest dies). To make the problem more complicated, the effects are not always immediate and they depend on the resistance matrix. The resistance of the specific pest type to a specific action is evaluated by the time after which the pest dies (the pest's immunity to the action is evaluated as infinite time). Pests can also have a maximum life-span after which they die regardless of the agent's actions. This maximum life span is called the *alive time*.

The problem can be further extended by introduction of deviations to the observed values of the pests' attributes or limiting the number of attributes the farmer agent can see.

In multi-agent systems, cooperation between agents is an important issue. The proposed environment offers several types of cooperation which can be taken into account. Some types of pests can attack several farms and the same action should be simultaneously executed in all of them to eliminate vermins. We can demand help-action execution by other agents to kill specific pests. Of course, such an extension makes the environment harder, because some synchronization mechanism is necessary.

[1]The proposed approach was also applied in other problems (Śnieżyński, 2013a; 2014); however, this was done before the formal model was formulated.

Algorithm 2. Strategy module algorithm (SM) for delayed rewards.

Require: $c_t \in C, per_t \in Per, s_t \in S$, $avm \in \{entr, disp\}$ representing a method used to measure action variation

Ensure: $act^* \in Act$ is the best action for given current state and perception calculated using c_t classifier

1: $\overrightarrow{x_t} := D(per_t, s_t)$; therefore $\overrightarrow{x_t}$ has a form of $(\overrightarrow{x^A_t}, \overrightarrow{x^A_{t-1}}, \ldots, \overrightarrow{x^A_{t-dt+1}})$
2: **if** $t < dt$ **then**
3: Fill attributes representing earlier time steps in $\overrightarrow{x_t}$ with unknown values
4: **end if**
5: Substitute attributes representing actions in $\overrightarrow{x_t}$ by unknown values
6: **for** $delay := 0$ **to** $dt - 1$ **do**
7: **for all** $act \in Act$ **do**
8: $\overrightarrow{x}^{act}_{delay} := (u^A_1, u^A_2 \ldots, u^A_{delay}, \overrightarrow{x^A_t}, \overrightarrow{x^A_{t-1}}, \ldots, \overrightarrow{x^A_{t-dt+1+delay}})$, where u^A_i means that all attributes have unknown values at position i
9: In $\overrightarrow{x}^{act}_{delay}$ set action at position $delay + 1$ equal to act
10: **end for**
11: **if** $avm = entr$ **then**
12: $\delta_{delay} := \sum_a c_t(\overrightarrow{x}^a_{delay}, good) \log_2 c_t(\overrightarrow{x}^a_{delay}, good)$ {Actions with certainty of $good$ category equal to zero are omitted in the sum}
13: **else**
14: $\delta_{delay} := \max_a c_t(\overrightarrow{x}^a_{delay}, good) - \min_a c_t(\overrightarrow{x}^a_{delay}, good)$
15: **end if**
16: **end for**
17: **if** $avm = entr$ **then**
18: $del^* := \arg\min_{delay} \delta_{delay}$
19: **else**
20: $del^* := \arg\max_{delay} \delta_{delay}$
21: **end if**
22: $act^* = \arg\max_{act} c_t(\overrightarrow{x}^{act}_{del^*}, good)$
23: **return** act^*

Another interesting aspect is the communication between farmer agents which allows them to exchange obtained knowledge, in this way improving their efficiency in fighting the pests. It is also a kind of cooperation.

Beside various configuration dimensions, the problem offers a flexibility in defining the goals and end-conditions as well. For example, the typical goal will be to kill as many pests as possible or to maintain the highest sum of time periods in which there were no pests on the fields. As for the end-condition, it can be defined in terms of time (given the time period or number of turns in case of discrete timing), results (reaching a specific number of kills) or other properties (e.g., a limited number of actions will enforce the farmer agent to optimize his/her choices).

The strategy learning model for this environment with immediate action results can be defined as follows. Pests are described by pest attributes $A^p = (a_1^p, a_2^p \dots a_n^p)$, where a_i^p has a domain D_i^p. The agent also observes if pests were killed last time, which is represented by $pk \in PK = \{y, n\}$. Hence, percepts are values of these attributes and pk: $Per = \times_i D_i^p \times PK$, $Act = \{act_1, act_2, \dots act_m\}$. If there are no delays in action results, then the state may represent the last action only: $S = Act$, and state update replaces the previous action: $SU(per, s, a) = a$. We use the QC approach, and therefore examples will be described by percepts (pest attributes) and action: $A = (A^p, Act)$, and categories $Cat = \{good, bad\}$ are used. As a result, experience $Exp = \times_i D_i^p \times Act \times Cat$.

Example generation is straightforward: EG is the defined by description, which simply omits pk: $D(per = (v_1, v_2, \dots, v_n, pk), s) = (v_1, v_2, \dots, v_n, s)$, and by category determination, which depends on pk:

$$CD(per = (v_1, v_2, \dots, v_n, pk), s) = \begin{cases} good & \text{if } pk = y, \\ bad & \text{if } pk = n. \end{cases} \quad (9)$$

SM applies c_t to per and all actions, and returns the action which gets the highest certainty of $good$ category. If action results are delayed, SM is defined in Algorithm 2. The Boltzmann selection method is used for exploration. The learning algorithm L is any supervised learning algorithm which gives, in the result, classifier c that, for a given example, returns certainties of categories, and works with unknown attribute values.

5. Experiments

The objective of experiments is to show that the proposed strategy learning model based on classification improves agents' effectiveness and to compare the learning speed with reinforcement learning. Using the farmer–pest problem, we are able to make experiments in various conditions: in chosen state-space sizes, with and without delays, using simple and complex dependencies between pest attributes and their types. We analyze how quickly agents improve their performance and show that various conditions favor different learning algorithms, and the proposed model may improve results faster than reinforcement learning.

Four learning agents take part in every experiment. They use reinforcement learning (SARSA) and the proposed strategy learning model, for which three types of learning algorithm (L) are tested: naïve Bayes, C4.5 or RIPPER. Their results are compared with those obtained by the fifth agent that executes random actions. Pests are described by four attributes. The number of actions is the same as that of pest types p: $Act = \{act_1, act_2, \dots act_p\}$, and every pest p^k of type k can be killed by act_k only after d turns.

Data were collected by running simulation software developed in Java (Weka's J48 and JRip implementations of C4.5 and RIPPER were used)[2]. Every experiment consists of 100 simulations. Every simulation consists of 20 consecutive games. $80/d$ pests may appear in every game. Knowledge of agents is preserved from game to game, although it is cleared between simulations. Supervised learning is executed between games while reinforcement learning is performed after every turn. In the experiments we check how the performance of agents, measured by eliminated pests, changes during simulation. The figures present the *efficiency (performance) of agents defined as means of numbers of pests eliminated by every agent in consecutive games*.

Agents applying supervised learning are implemented using the proposed model (and Algorithm 1). Algorithm 2 is used for action choice in the strategy module (SM).

Three experiments with various configuration settings were performed. Action variation was measured using dispersion or entropy. Action a_t eliminated the pest p^t after delay $d > 1$. In all experiments we used a time window of size $dt = 8$.

Tuning learning algorithms' parameters. To choose the best configurations for all algorithms and make the comparison fair, the learning algorithm parameters were tuned using the hill-climbing algorithm. For every set of parameters considered, its score was calculated as the average efficiency of the learning agent in the last round in simulation executed 100 times. Simulation used the simple environment described below in Experiment 1, but d was equal to four. The following parameter values obtained the best score for SARSA: $\lambda = 0.9, \gamma =$

[2]The software and settings necessary to run the experiments are available from the author on an e-mail request.

$0.8, \beta = 0.3, \tau = 0.05$. For supervised learning algorithms, only τ had to be tuned: for Naïve Bayes $\tau = 0.15$, for C4.5 $\tau = 0.15$ and for RIPPER $\tau = 0.1$. These values were used in all experiments.

Experiment 1: Action variation measures comparison. This experiment is designed to compare two action variation measures used in the strategy module (SM): entropy and dispersion. We use two environments: simple and complex. The time window size was equal to the delay: $dt = d = 8$.

In the *simple environment*, pests are described by four attributes with discrete values. Every pest p^k of type k has its unique values of all attributes a_1, a_2, a_3, a_4:

$$a_i(p^k) = 10k, \quad i = 1, \ldots, 4, \quad k = 1, \ldots, 8. \qquad (10)$$

It is possible to recognize every type using any of the attributes. In the *complex environment*, attributes with domains of size two or three are used. The attribute values are presented in Table 1. As we can see, distributions are chosen in such a way that no single attribute can be used to recognize the pest type. Tests on several attributes are necessary. Two configurations of the complex environment are used: with four pests (Type 1–4) and for eight pests. In both $d = dt = 8$.

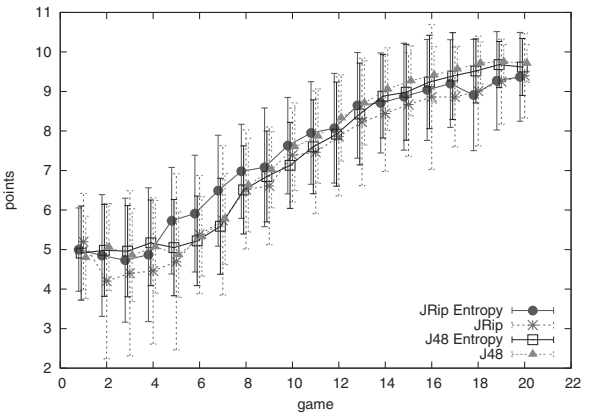
Table 1. Pest's attribute values for the complex environment.

Pest	size		legsno		speed			jump		
type	40	10	40	30	40	20	10	40	20	10
1	x		x		x			x		
2	x		x		x				x	
3		x	x		x			x		
4		x	x		x				x	
5	x			x	x			x		
6	x			x	x				x	
7		x		x	x			x		
8		x		x	x				x	

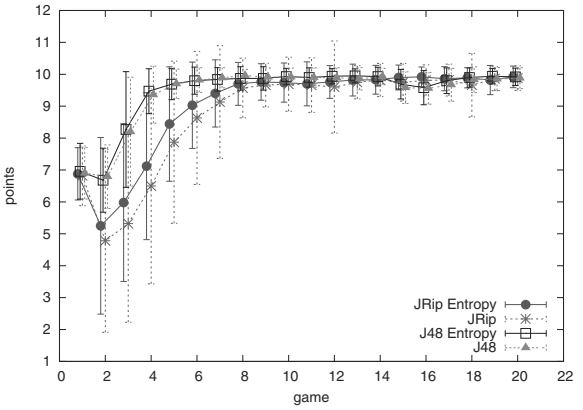
Two agents use the dispersion measure (JRip, J48) and two use entropy (JRip Entropy, J48 Entropy).

The results are presented in Fig. 2. One can notice that for four pest types (b) the learning process is faster than for eight pest types (a) and (b). The reason is that there are fewer possible pest descriptions and fewer possible actions to try (equal to the number of pest types).

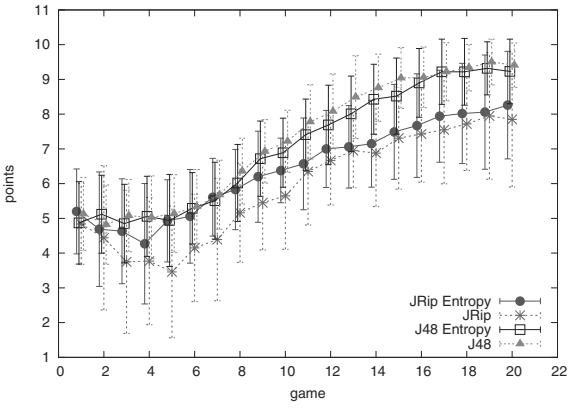
In all configurations, the JRip-Entropy agent learns faster than JRip-Dispersion. In the fifth game the difference is statistically significant (using the t-test, at $p < 0.05$). J48 yield similar results for both measures. Therefore, in the following experiments the action variation was measured using entropy.



(a)



(b)



(c)

Fig. 2. Means of numbers of pests eliminated by every agent in consecutive games in Experiment 1 for simple configuration (a), complex configuration and four pests (b) and eight pests (c).

Experiment 2: Simple environment. In this experiment simple environment described above was used. To compare learning algorithms in various conditions, three values of d were used in the experiments: 2, 4 and 8.

The results are presented in Fig. 3 (a)–(c). The random agent and NB agent have poor results. The

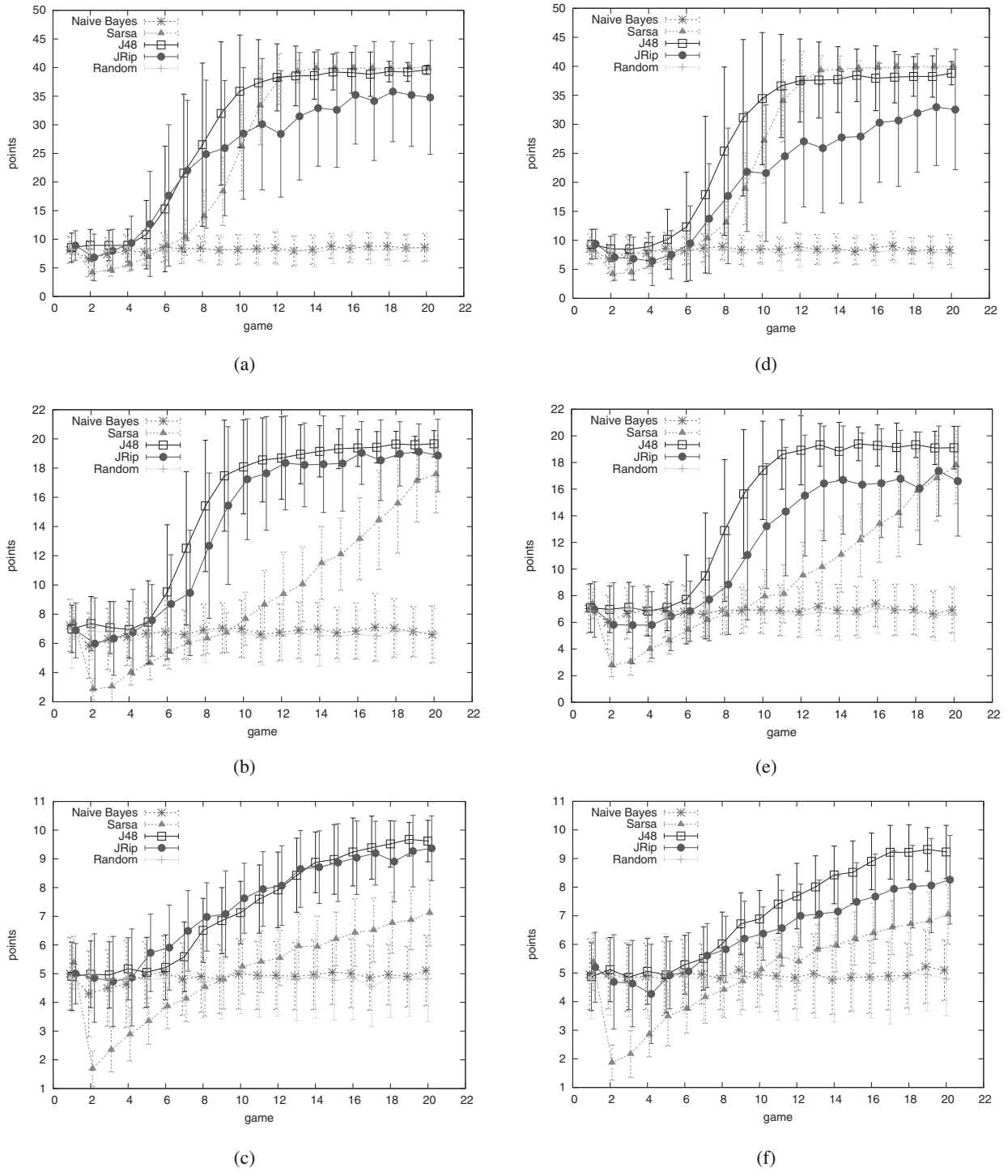


Fig. 3. Means of numbers of pests eliminated by every agent in consecutive games in Experiment 2 for $d = 2$ (a), $d = 4$ (b) and $d = 8$ (c), and in Experiment 3 for $d = 2$ (e), $d = 4$ (f) and $d = 8$ (g).

explanation for NB is simple. This model is not able to take into account the dependency between state attributes and action attributes, which is crucial in the proposed model with delays. The remaining learning agents perform much better. If d is larger, the learning process is slower. For SARSA, the difference is especially noticeable. For $d = 2$ (a), SARSA is almost as fast as C4.5 and outperforms RIPPER. For $d = 4$ (b) it is slower than C4.5 and RIPPER, but finally the results are still good. In the case of $d = 8$ (c), SARSA is not able to achieve

as good results as the two supervised algorithms in the observed time and the difference is statistically significant (using the t-test, at $p < 0.05$). The state space is too large. At the beginning (first several games), SARSA is worse than the random choice and NB. In the second game, the difference is statistically significant (using the t-test, at $p < 0.05$). This shows that reinforcement learning needs more trials to avoid a false local optimum. In this experiment in the last game RIPPER and C4.5 are always better than the random choice and NB, and the difference is significant (using the t-test, at $p < 0.05$).

Experiment 3: Complex environment. In this experiment the complex environment from Experiment 1 was used. Three values of d were used in experiments: 2, 4 and 8. Action variation was measured using entropy.

The results for the complex environment are presented in Fig. 3(d)–(f). NB is not working again. The less d, the better results can be finally achieved. For $d = 2$ (d), C4.5 and SARSA learn quickly and outperform other agents (the result is statistically significant at $p < 0.05$). For $d = 8$ (f), C4.5 is statistically better than others (at $p < 0.05$) and RIPPER is better than SARSA. Again, during the initial games, one may observe very poor results of SARSA (even worse than the random choice).

Comparing the results for both environments, we can observe that a change in the environment complexity affects the RIPPER algorithm more than SARSA and C4.5. For the same d, SARSA has similar learning speed in both cases. This is caused by the tabular Q-function representation, which has the same performance for both cases. C4.5 is only slightly slower for the complex environment. This classifier is able to separate classes well even in more complex environment. The RIPPER classifier has a worse accuracy for the second one.

6. Conclusion and further research

In this paper we present a model of agent strategy generation using supervised learning. We compared the performance of the reinforcement and supervised learning algorithms: SARSA, naïve Bayes, C4.5 and RIPPER in the farmer–pest problem, which is a scalable multi-dimensional problem domain for testing agent learning algorithms. This environment provides numerous configurable dimensions, which enables preparation of different testing conditions.

The experimental results show that supervised learning provides efficiency improvements faster than reinforcement learning. When actions have delayed results, naïve Bayes cannot be used for learning because of the attribute independence assumptions. In such conditions, SARSA has poor results during several initial games (worse than the random choice).

For simple environments, in which any attribute value allows choosing the right action, every learning algorithm produces fast improvements, unless the reward delay d is too large (it is a problem especially for SARSA). If the environment is more difficult, when it happens that the agent should take into account values of several attributes to choose an appropriate action, and d is not small ($d \geqslant 4$), the C4.5 and RIPPER supervised learning algorithms perform better than SARSA.

It should be noted that, when the knowledge learned by agents should be interpreted by humans, supervised learning algorithms with symbolic knowledge representation should be preferred (if this gives acceptable results). Knowledge stored during the learning process can have a form which makes it possible to be interpreted by a human. Especially decision rules or trees are good choices.

Future work will be concentrated on the execution of experiments with more algorithms. Another aspect of the work will be the extension of testing environment to cover cooperation between agents and delays in action results. Symbolic knowledge representation makes it possible to take into account complex dependencies between environment attributes and decisions to be made. Next, we are planning other applications like resource allocation (Cetnarowicz and Drezewski, 2010).

Acknowledgment

The research presented in this paper was supported by the Polish Ministry of Science and Higher Education under the AGH University of Science and Technology grant no. 11.11.230.124.

References

Airiau, S., Padham, L., Sardina, S. and Sen, S. (2008). Incorporating learning in BDI agents, *Proceedings of the ALAMAS+ALAg Workshop, Estoril, Portugal*.

Barrett, S., Stone, P., Kraus, S. and Rosenfeld, A. (2012). Learning teammate models for ad hoc teamwork, *AAMAS Adaptive Learning Agents (ALA) Workshop, Valencia, Spain*.

Bazzan, A., Peleteiro, A. and Burguillo, J. (2011). Learning to cooperate in the iterated prisoners dilemma by means of social attachments, *Journal of the Brazilian Computer Society* **17**(3): 163–174.

Bellman, R. (1957). *Dynamic Programming*, A Rand Corporation Research Study, Princeton University Press, Princeton, NJ.

Cetnarowicz, K. and Drezewski, R. (2010). Maintaining functional integrity in multi-agent systems for resource allocation, *Computing and Informatics* **29**(6): 947–973.

Cohen, W.W. (1995). Fast effective rule induction, *Proceedings of the 12th International Conference on Machine Learning (ICML'95), Tahoe City, CA, USA*, pp. 115–123.

Dietterich, T.G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition, *Journal of Artificial Intelligence Research* **13**: 227–303.

Gehrke, J.D. and Wojtusiak, J. (2008). Traffic prediction for agent route planning, *in* M. Bubak *et al.* (Eds.), *Computational Science—ICCS 2008, Part III*, Lecture Notes Computer Science, Vol. 5103, Springer, Berlin/Heidelberg, pp. 692–701.

Hernandez-Leal, P., Munoz de Cote, E. and Sucar, L.E. (2013). Learning against non-stationary opponents, *Workshop on Adaptive Learning Agents, Saint Paul, MN, USA*.

Kaelbling, L.P., Littman, M.L. and Moore, A.W. (1996). Reinforcement learning: A survey, *Journal of Artificial Intelligence Research* **4**: 237–285.

Kazakov, D. and Kudenko, D. (2001). Machine learning and inductive logic programming for multi-agent systems, *in* M. Luck *et al.* (Eds.), *Multi-Agent Systems and Applications*, Springer, Berlin/Heidelberg, pp. 246–270.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching, *Machine Learning* **8**(3–4): 293–321.

Panait, L. and Luke, S. (2005). Cooperative multi-agent learning: The state of the art, *Autonomous Agents and Multi-Agent Systems* **11**(3): 387–434.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Francisco, CA.

Rao, A.S. and Georgeff, M.P. (1991). Modeling rational agents within a BDI-architecture, *in* J. Allen, R. Fikes and E. Sandewall (Eds.), *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann: San Mateo, CA, pp. 473–484.

Rummery, G.A. and Niranjan, M. (1994). On-line q-learning using connectionist systems, *Technical report*, Cambridge University, Cambridge.

Russell, S.J. and Zimdars, A. (2003). Q-decomposition for reinforcement learning agents, *Proceedings of the 20th International Conference on Machine Learning (ICML-2003), Washington, DC, USA*, pp. 656–663.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*, 3rd Edn., Prentice-Hall, Upper Saddle River, NJ.

Sen, S. and Weiss, G. (1999). *Learning in Multiagent Systems*, MIT Press, Cambridge, MA, pp. 259–298.

Shoham, Y., Powers, R. and Grenager, T. (2003). Multi-agent reinforcement learning: A critical survey, *Technical report*, Stanford University, Stanford, CA.

Singh, D., Sardina, S., Padgham, L. and Airiau, S. (2010). Learning context conditions for BDI plan selection, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, Toronto, Canada*, pp. 325–332.

Śnieżyński, B. (2013a). Agent strategy generation by rule induction, *Computing and Informatics* **32**(5): 1055–1078.

Śnieżyński, B. (2013b). Comparison of reinforcement and supervised learning methods in farmer–pest problem with delayed rewards, *in* C. Badica, N.T. Nguyen and M. Brezovan (Eds.), *Computational Collective Intelligence*, Lecture Notes in Computer Science, Vol. 8083, Springer, Berlin/Heidelberg, pp. 399–408.

Śnieżyński, B. (2014). Agent-based adaptation system for service-oriented architectures using supervised learning, *Procedia Computer Science* **29**: 1057–1067.

Śnieżyński, B. and Dajda, J. (2013). Comparison of strategy learning methods in farmer–pest problem for various complexity environments without delays, *Journal of Computational Science* **4**(3): 144 – 151.

Śnieżyński, B. and Kozlak, J. (2005). Learning in a multi-agent approach to a fish bank game, *in* M. Pchouek, P. Petta and L.Z. Varga (Eds.), *Multi-Agent Systems and Applications IV*, Lecture Notes in Computer Science, Vol. 3690, Springer, Berlin/Heidelberg, pp. 568–571.

Śnieżyński, B., Wojcik, W., Gehrke, J.D. and Wojtusiak, J. (2010). Combining rule induction and reinforcement learning: An agent-based vehicle routing, *Proceedings of the International Conference on Machine Learning and Applications, Washington, DC, USA*, pp. 851–856.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*, The MIT Press, Cambridge, MA.

Sutton, R.S. (1990). Integrated architecture for learning, planning, and reacting based on approximating dynamic programming, *Proceedings of the 7th International Conference on Machine Learning, Austin, TX, USA*, pp. 216–224.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents, *Proceedings of the 10th International Conference on Machine Learning, Amherst, MA, USA*, pp. 330–337.

Tuyls, K. and Weiss, G. (2012). Multiagent learning: Basics, challenges, and prospects, *AI Magazine* **33**(3): 41–52.

Watkins, C.J.C.H. (1989). *Learning from Delayed Rewards*, Ph.D. thesis, King's College, Cambridge.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*, 2nd Edn., Wiley Publishing, Chichester.

Zhang, W. and Dietterich, T.G. (1995). A reinforcement learning approach to job-shop scheduling, *Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Canada*, pp. 1114–1120.

Bartłomiej Śnieżyński received his Ph.D. degree in computer science in 2004 from the AGH University of Science and Technology in Cracow, Poland. In 2004 he worked as a postdoctoral fellow under the supervision of Prof. 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. He is a member of the Polish Information Processing Society (PTI) and the Polish Artificial Intelligence Society (PSSI).

Received: 14 July 2014
Revised: 8 January 2015