Improving Evolutionary Learning of Cooperative Behavior by Including Accountability of Strategy Components

Jörg Denzinger and Sean Ennis

Department of Computer Science, University of Calgary, Canada
{denzinge,ennis}@cpsc.ucalgary.ca

Abstract. We present an improvement to evolutionary learning of cooperative behavior which incorporates some accountability measure for strategy components into the evolutionary learning process. Our evolutionary approach is based on evolving sets of prototypical situation-action pairs (strategies) that together, with the nearest-neighbor rule, represent the decision making of our agents. The basic idea of our improvement is to collect data for each pair showing the results of its applications. We then choose those pairs in the parent strategies that had positive results for the construction of new sets of pairs for our strategies. Our experiments within the OLEMAS system show that the incorporation of accountability results in substantial improvements of both on- and off-line learning when compared to the basic evolutionary approach. In nearly all experiments, either the agent teams required less learning time or found better strategies. In many cases both were observed.

1 Introduction

Having the ability to learn is often viewed as a very important feature of an agent in a multi-agent system. Adapting to new (resp. slightly changed) environments, dealing with new agents, or relieving the human developer from having to develop all details of a cooperation concept are just a few of the consequences we hope for when agents are able to learn. Naturally, different agent architectures require different learning methods, but a lot of the research done on learning in multi-agent systems has focused on rather reactive agents. Two general ideas for such learning have surfaced: reinforcement learning (see [11], [10], or [6]) and evolutionary learning.

In this paper, we focus on evolutionary learning of cooperative behavior. Evolutionary learning (EL) concentrates on finding whole strategies (in contrast to the reinforcement approach that focuses on all possible situations and all possible actions in these situations). Working on a pool of strategies, evolutionary techniques are used to generate new strategies (mostly out of the better strategies in the pool) that hopefully combine positive aspects from the parents. Over time, strategies evolve that come progressively closer to achieving the intended behavior (for examples, see [8], [5], or [3]). The evolved strategies are usually

M. Schillo et al. (Eds.): MATES 2003, LNAI 2831, pp. 205–216, 2003.
© Springer-Verlag Berlin Heidelberg 2003

much more compact than the weight matrices or graphs of reinforcement learning. Due to this, substantially fewer experiences are necessary to evolve working strategies when compared to reinforcement learning (even if we combine the experiences obtained by every strategy tried out in the evolutionary process). On the negative side, so far, even the successful strategies often include elements that are not needed (resp. wrong) and are therefore simply not used in the solution. In addition, since experiences are attributed to whole strategies and not individual actions (or small action sequences) the learning is less focused than it is in reinforcement learning (which sometimes can be a positive asset, but can also hinder the progress of learning).

In this paper, we present an improvement of the evolutionary learning method of [3] and [4] that integrates an accountability aspect to deal with the problem mentioned above, namely statistics about single pairs of situations and actions, into the evolutionary learning process. More precisely, our learning approach is based on prototypical situation-action pairs (SAPs) and the nearest-neighbor rule as the agent architecture. A strategy of an agent is then a set of such prototypical SAPs. In the basic version, the fitness of a strategy is obtained by measuring how near a strategy comes to solving the given problem during simulations of the whole multi-agent system and its environment.

Our improvement idea is to not only compute the fitness of a strategy out of the simulations, but also statistics about the use and consequent success of the SAPs in the strategy. We then use these statistics to influence the application of the genetic operators that generate new strategies. In the basic version of the learning algorithm, after selecting parent strategies based on their fitness (and some random decisions), picking SAPs to either be included or excluded from the new strategy is done purely at random. In our improved version, this picking is now performed based on the statistics about the pairs (and again, some random decisions), thus repeating the selection idea of the strategy level. The general idea we use is that SAPs whose application often resulted in better situations should be selected with a higher probability than pairs that generally did not improve the situation of agent and agent team and the pairs that made the situation worse.

We implemented this improvement into the OLEMAS system (see [4] and [2]). Our experiments in the area of Pursuit Games show that the new version including accountability aspects clearly outperforms the basic version of OLEMAS for almost all game variants and for both usage in off-line and on-line learning. Outperforms in this context means that either less generations of the GA are needed to find a successful strategy (thus also reducing the learning time) or that the found strategies are better than those found by the basic version (i.e. less actions are performed by the agent team until success) or both.

2 Learning with SAPs

In this section, we present the method of evolutionary learning of [3] and [4], as realized in the OLEMAS system (**O**n-**L**ine **E**volution of **M**ulti-**A**gent **S**ystems),

that is the basis for our work. We start by presenting the agent architecture used, followed by the GA we use for learning, and finally we briefly discuss the use of this basic learning method for off- and on-line learning.

2.1 Agent Architecture: SAPs and NNR

Very abstractly, an agent $\mathcal{A}g$ can be described by a triple $\mathcal{A}g = (Sit,Act,Dat)$, where Sit is the set of situations $\mathcal{A}g$ can be in, Act is the set of actions $\mathcal{A}g$ can perform, and Dat is the set of possible values of the internal data areas of $\mathcal{A}g$. $\mathcal{A}g$ then realizes a function $f_{\mathcal{A}g}$: $Sit \times Dat \rightarrow Act$. In reactive agents, the emphasis of $f_{\mathcal{A}g}$ is mainly on Sit.

In our agent architecture, $f_{\mathcal{A}g}$ bases its decisions on a set of prototypical situation-action pairs, its strategy, that are part of an area in Dat. As the name suggests, an SAP contains an element of Sit[1] and an action from Act. For determining what action to perform in a situation s, the agent computes the similarity (resp. distance) of all situations in its strategy and s, and performs the action of the SAP whose situation is most similar to s (i.e. it applies the nearest-neighbor rule, NNR). Naturally, there usually are different possible definitions for similarity. Also, we have to describe situations in such a way that the definition of a sensible similarity measure is possible.

The behavior B of an agent $\mathcal{A}g$ starting with a situation s_0 can be described as a sequence $B(\mathcal{A}g,s_0) = s_0,\text{sap}_1,s_1,...,s_{i-1}, \text{sap}_i,s_i,...$, where sap_j is an element of its strategy and the action associated with it leads $\mathcal{A}g$ from situation s_{j-1} to s_j. Naturally, if there are other agents in the system, then s_j also depends on the actions they perform in s_{j-1}.

2.2 The Basic GA for Learning

Our evolutionary learning method is based on a Genetic Algorithm for sets (since we use sets of SAPs as strategies). This means that for learning we always consider a set of strategies. New strategies are generated out of old strategies by applying so-called Genetic Operators, which in our case are Crossover and Mutation. The initial set of strategies (initial population) is generated randomly (although in [2] we presented a variant that makes use of previous knowledge), i.e. by generating random SAPs.

Crossover requires two parent strategies, $\text{st}_1 = \{\text{sap}_{11},..., \text{sap}_{1n}\}$ and $\text{st}_2 = \{\text{sap}_{21},...,\text{sap}_{2m}\}$, and generates a new strategy st_{new} by picking randomly the needed number of SAPs out of $\text{st}_1 \cup \text{st}_2$ (without duplicates). *Mutation* requires only one parent st_1 and in order to generate a st_{new}, it allows for three possibilities, namely deleting a random SAP of st_1, i.e. $\text{st}_{new} = \text{st}_1 - \text{sap}_{1j}$, $j \in \{1, ..., n\}$, generating a SAP sap randomly and adding it to st_1 (provided that st_1 does not already have the maximal allowed number of SAPs), i.e. $\text{st}_{new} = \text{st}_1 \cup \text{sap}$, or exchanging a SAP in st_1 by a randomly generated one (sap), i.e. $\text{st}_{new} = \text{st}_1 -$

[1] Situations might be extended to also contain data from the current value of Dat of the agent.

$sap_{1j} \cup$ sap (which combines the other two possibilities; again, duplicates are not allowed). We have organized generating new strategies into so-called "generations", i.e. we generate a given number l of new strategies and then form a new generation by deleting the l worst strategies from the old generation and adding to it the l newly generated ones.

The last sentence already referred to another basic requirement of GAs: the ability to measure the quality, or the *fitness*, of the individuals in a population. The fitness is not only needed to delete strategies, it also is a key component in selecting the parent strategies, although it is combined with a random influence. There are many different ways to combine fitness and randomness, and in OLEMAS we have chosen a variant in which the probability of a strategy for being selected as parent is proportional to its fitness.

For measuring the fitness of an individual strategy (in fact, for the strategies of all agents of a team) we measure the success it produces in every step of its application (for a given limited number of steps, either in the real world or in a simulation of it), except if the strategy is totally successful, in which case the fitness is just the number of steps (length of the action sequence) needed to fulfill the given goal. More precisely, since the success obviously is related to the application, we need a function $\delta : Sit \rightarrow \mathbb{N}$ measuring how far away a situation is from success. Then we take the behavior of the agent $\mathcal{A}g$ employing the strategy from the start situation s_0, i.e. $B(\mathcal{A}g,s_0)$, and sum up $\delta(s_j)$ for all s_j in $B(\mathcal{A}g,s_0)$.

If the agents have to deal with effects out of their control (for example, random effects or other agents that cannot be predicted) then, starting from s_0, different behaviors can be observed in different runs. The fitness is then computed as the sum of the elemental fitnesses generated by each of the observed behaviors in a given number of runs of the strategy.

If we want to learn strategies for several agents, then an individual in our Genetic Algorithm contains an individual strategy for each of the agents. The fitness of an individual is still the summed up $\delta(s_j)$ for all situations in the behavior of one agent, since each s_j is the consequence of the actions of all agents taken in the previous situation.

2.3 Offline and Online Learning

With regard to learning, one very often finds the distinction between on-line and off-line learning with the later meaning that learning and applying the learned knowledge are separated in different phases. In contrast, the former means that the learning, and the application of what is learned are interleaved so that problems like when to learn, or what to do when learning is not finished, have to be solved. For learning cooperative behavior for a team of agents that has to learn to solve a certain problem/task, pure off-line would mean that learning takes place first and then the agent team has to perform a run that solves the task without doing any more learning. Consequently, an on-line learning agent or agent team will also learn during the run. These are only the extremes, however, and a lot of combinations of them are also possible.

Our evolutionary learning, as described in the previous subsection, can be used for both, off- and on-line learning. If the start situation s_0 is the situation the team has to start from when solving the given task, and if the number of steps allowed for the fitness evaluation is the number of steps allowed for the task, then our learning approach can be used for off-line learning. This is to say that at the end of the learning we will have a strategy (for each learning agent) that then will be used for solving the task (see [3]). In [4], we presented a way to use our approach for on-line learning as well by introducing a special action "learn" for the agents. Its application leads to executing our learning GA for a rather small number of steps, starting from a situation that the agent thinks it will be in after executing "learn", and using models of the other agents to predict their behavior. So, in on-line learning the individuals in our Genetic Algorithm contain only the one strategy for the agent executing "learn", even if several agents are doing on-line learning. With regard to learning, every agent "is on its own". As we will see in Section 5, our improvement of the basic GA will improve both the use for off- and on-line learning.

3 Adding Accountability of SAPs

One of the big advantages of evolutionary learning as described in the last section is that it partially avoids having to solve the credit assignment problem (i.e. having to determine how much a particular action contributed to the success of an action sequence, a basic problem in reinforcement learning). Since a fitness is computed for a whole strategy (a posteriori), it is not necessary to develop a sophisticated mechanism for deciding a priori how much a particular action and its immediate outcome will be responsible for the final outcome of an action sequence (although our fitness uses some crude estimation of the success of each action in a sequence to compare strategies that were not totally successful). In the case of off-line learning for several agents, we also do not have to decide which agent contributed how well to the team effort. Unfortunately, this means that very good actions (or good strategies for other agents) can compensate for not-so good ones, as long as we achieve success in the end. Even worse, with our particular agent architecture we can have useless SAPs in very successful strategies and their uselessness is not detected due to the fact that they were never responsible for an action taken.

Our idea for improving our evolutionary approach is to add some accountability to the prototypical SAPs of a strategy (not to all possible SAPs) and to use this accountability to influence the Genetic Operators. This influence will be in such a way that "bad" or useless SAPs are less likely to appear in offspring of strategies (only less likely, because together with another set of SAPs they might be valuable; see our experimental evaluation in Section 5 that compares this approach to an approach were the SAP selection is purely based on the observed quality of the SAPs). Each action occurring in the observed behavior of an agent will provide feedback, and this feedback will be used to determine good, indifferent, bad, and unused SAPs. This idea combines the a posteriori

evaluation of whole strategies, provided by the basic evolutionary learning, with the advantage of accountability of strategy parts.

More precisely, we extend a SAP in a strategy by a so-called statistic-tuple stat(sap) = (use-nr, good, bad, indiff). Whenever a new strategy st is created, the statistic-tuples of all its SAPs $\{sap_1^{st},..., sap_n^{st}\}$ are initialized to (0,0,0,0). For each run of st as agent $\mathcal{A}g$ (starting with a situation s_0), we use the resulting behavior $B(\mathcal{A}g,s_0) = s_0,sap_1,s_1,...,s_{i-1},sap_i, s_i$ to update the statistic-tuples in st as follows:
For all s_{k-1},sap_k,s_k in $B(\mathcal{A}g,s_0)$, $1 \leq k \leq i$:

$$stat(sap_j^{st}) = \begin{cases} stat(sap_j^{st}), \text{if } sap_j^{st} \neq sap_k \\ (\text{use-nr} + 1, \text{good} + 1, \text{bad, indiff}), \text{if } \delta(s_{k-1}) > \delta(s_k) \\ (\text{use-nr} + 1, \text{good, bad, indiff} + 1), \text{if } \delta(s_{k-1}) = \delta(s_k) \\ (\text{use-nr} + 1, \text{good, bad} + 1, \text{indiff}), \text{if } \delta(s_{k-1}) < \delta(s_k) \end{cases}$$

for all $j \in \{1, ..., n\}$. Naturally, the last 3 cases require that $sap_j^{st} = sap_k$. Note that we have chosen to reuse the δ-function we already use in the fitness function. Obviously, other functions can also be used. This is also the case for several other decisions we already have made and will make in the following. So, we judge the impact that each use of a SAP has, but we only use three categories, namely positive, negative, and indifferent impact. Note that, by using δ, the impact of a SAP is measured relative to the decisions the other agents did make, since they also influence what the successor situation of a situation is, but for learning of cooperative behavior, this should obviously be the case!

The statistic-tuples of SAPs are then used to modify our Genetic Operators. For Crossover, we have strategies $st_1 = \{sap_{11},..., sap_{1n}\}$ and $st_2 = \{sap_{21},...,sap_{2m}\}$. We take $st_1 \cup st_2$ and divide its SAPs into three pools:

$$\begin{aligned} \text{good_pool} &= \{sap \in st_1 \cup st_2 \mid \text{use-nr} > 0 \text{ and good - bad} > \text{good_min}\} \\ \text{indiff_pool} &= \{sap \in st_1 \cup st_2 \mid \text{use-nr} > 0 \\ &\quad \text{and good_min} \geq \text{good - bad} \geq \text{bad_max}\} \\ \text{bad_pool} &= \{sap \in st_1 \cup st_2 \mid \text{use-nr} = 0 \text{ or use-nr} > 0 \\ &\quad \text{and bad_max} > \text{good - bad}\} \end{aligned}$$

Here, good_min and bad_max are parameters that allow us better control over which SAPs go into which pool. One obvious value for both of them is 0, which results in having in good_pool all SAPs that more often resulted in better situations than worse. Then indiff_pool is usually rather small. By having good_min > 0 and bad_max < 0, we can broaden indiff_pool a little bit and make it tougher to get into good_pool. The same three pools can also be defined for only one strategy, which we have in case of Mutation.

For generating a new strategy st by Crossover, we use the pools as follows. We use two parameters p_{good} and p_{indiff}, $p_{good} + p_{indiff} \leq 1$, that define the percentages for the SAPs taken from the pools. If st is supposed to have q SAPs, then we randomly select

– $\lceil p_{good} \times q \rceil$ SAPs out of good_pool,

- $\lceil p_{indiff} \times q \rceil$ SAPs out of good_pool ∪ indiff_pool, and
- $q - \lceil p_{good} \times q \rceil$ - $\lceil p_{indiff} \times q \rceil$ SAPs out of good_pool ∪ indiff_pool ∪ bad_pool.

This means that for each of the q SAPs needed, SAPs in good_pool have a chance to be selected, while SAPs in the other pools are eligible for less "positions" in the new strategy. Note that in the case of a pool containing less SAPs than needed, all SAPs of this pool are selected and the remaining allotment for this pool will be selected out of the next lower pool.

The intent of having Mutation is to add new SAPs to the gene pool (i.e. the SAPs occurring in any strategy of the population). Therefore accountability aspects are not so important for Mutation. So, we still use the three kinds of Mutation we discussed in Section 2.2, but we can also add variants of the delete and exchange Mutations, in which we delete/exchange not a random SAP, but a random SAP of bad_pool.

After having seen the usage of statistic-tuples, one might ask if it is really necessary to initialize the tuples for the SAPs in a new strategy to a zero-vector. Why not inherit the statistics from the parent? Due to using the nearest-neighbor rule for action selection, the statistic-tuples have a certain dependency on the other SAPs in a strategy, especially with respect to the bad and indiff numbers in the tuple. If a particular SAP is put into a new strategy, in situations it previously was responsible for the action taken now another SAP might become responsible for the action (if its situation is more similar). In addition, if we learn strategies for several agents in one individual, the statistic tuples reflect accountability of actions with respect to the actions of the other agents (as already mentioned). In a new individual, some of the other agents will act differently. Therefore SAPs should not inherit their statistics from their parents.

4 Pursuit Games and the OLEMAS System

In order to test our improvement of evolutionary learning with accountability of SAPs, we integrated our approach into the OLEMAS system (see [4] and [2]). OLEMAS presents many variants of Pursuit Games as an application domain.

4.1 Pursuit Games

Pursuit Games were first introduced in multi-agent systems in [1]. Since then, many variants of them have been introduced, see [3] for a list of features that can be varied. The general idea of a Pursuit Game is to have a group of hunter agents and one or several prey agents that move on a playing field consisting of connected grids. The goal of the game is to have the hunters catch the prey within a given limit of so-called turns, where the "catch" can be defined in several ways (see Section 5).

Due to the many features that can be varied –like number of agents involved, obstacles, possible actions agents can take, their size, shape and speed, many possible random influences– very different game variants can be defined, some

favoring the hunters, some the prey. Developing good strategies for the hunter agents that achieve the necessary cooperative behavior to win the game, is, even for a single variant, not always easy. However, given the enormous number of possible variants, letting the hunters learn their strategies becomes a must.

4.2 Instantiating Our Approach for OLEMAS

In OLEMAS, a situation is described by a vector providing the coordinates of all visible agents relative to the agent for which the situation is described, in a fixed agent order. In addition, for each of these agents we also provide in the vector their type and orientation. The set of actions of an agent can contain, in addition to moves in the different directions and staying put, turns and for on-line learning agents "learn". Associated with each action of an agent is the number of turns this agent needs to perform the action.

For measuring the similarity of two situations, we sum up the squares of the difference in numbers of the corresponding coordinate fields and the orientation fields of the two situation vectors. The function δ, that is used for both, fitness computation and updating the statistic-tuples, is the sum of the Manhattan distances between each hunter agent and each prey agent.

5 Experimental Evaluation

We have performed experimental series with a number of variants of Pursuits Games to evaluate the general usefulness of our improvement. Due to lack of space, we cannot present all experiments and therefore selected the most different ones with regard to the different features. We tested both the on- and off-line versions of OLEMAS. In our experiments, we examined the (average) time needed for learning and the quality of the found solution. The later is expressed by the (average) number of steps (turns) needed by the hunter agents to catch the prey and by the success rate, i.e. the percentage of system runs that have a positive result (i.e. hunters catching the prey within the given limit of steps). Naturally, the success rate is only of interest if either the game variants include random factors or we perform on-line learning.

In addition to comparing the base learning algorithm of [3], resp. [4], with our improvement, we also compared our improvement described in Section 3 with an obvious variant of our general idea of including accountability into EL. This variant is selecting the SAPs from the parent strategies totally controlled by the success of the SAPs with the parents, without the additional random influences that we proposed in Section 3. As we will see, this variant, that we call *pure success-based*, is already better than the base algorithm and sometimes also better than what we proposed in Section 3, which in the following we will call *success-influenced*. The overall performance of the success-influenced version is better than the pure success-based one (see later).

The general setting of all experiments is that we defined a game variant, the basic parameters of our GA and the basic learning parameters, and between the

Table 1. Experimental results for off-line learning, times in minutes

Variant	Base algorithm			Pure success-based			Success-influenced		
	time	steps	success	time	steps	success	time	steps	success
1	**555.00**	**99**	-	-	-	-	-	-	-
2	1144.00	77	50%	1071.00	82	**60%**	**971.00**	**51**	**60%**
3	20.18	73	60%	20.70	37	70%	**9.18**	**30**	**85%**
4	34.25	51	70%	17.18	47	**80%**	**12.07**	**45**	**80%**
5	-	-	-	544.00	326	**20%**	**375.00**	**272**	15%
6	0.38	73.5	**100%**	**0.18**	**59.4**	**100%**	0.23	59.9	**100%**
7	375.00	107	30%	281.00	**47**	**55%**	**244.00**	51	**55%**
8	5.18	190	-	**4.85**	**109**	-	5.03	177	-
9	82.07	43	-	79.05	44	-	**74.00**	**31**	-
10	69.10	71	-	41.18	69	-	**40.67**	**57**	-
11	135.18	117	-	82.07	91	-	**60.00**	**72**	-

Table 2. Experimental results for on-line learning, times in minutes

Variant	Base algorithm			Pure success-based			Success-influenced		
	time	steps	success	time	steps	success	time	steps	success
1	**6.65**	**153.4**	**90%**	8.58	232.9	80%	7.38	197.4	**90%**
2	448.00	97.3	60%	464.00	79.2	**70%**	**419.00**	**65.4**	**70%**
3	0.36	59.8	90%	0.33	50.4	**100%**	**0.24**	**48.1**	**100%**
4	0.32	54.1	95%	0.30	50.4	**100%**	**0.28**	**41.5**	**100%**
5	177.12	61.3	65%	105.00	71.9	85%	**92.23**	**60.9**	**90%**
6	0.29	35	**100%**	**0.19**	**29.4**	**100%**	0.20	32.4	**100%**
7	132.08	72.5	55%	111.23	51.7	70%	**103.98**	**40.1**	**80%**
8	2.70	214.5	**80%**	**2.34**	**199.3**	**80%**	2.37	201.3	75%
9	5.78	141.9	90%	**2.59**	68.8	**100%**	2.90	**65.5**	**100%**
10	14.05	73.5	55%	11.08	61.8	65%	**10.68**	**39.7**	**70%**
11	7.00	171.3	45%	5.13	116.7	55%	**3.53**	**64.9**	**90%**

three tested learning algorithm variants the only differences are whether and how the statistical data of the SAPs was used in the Genetic Operators. For the success-influenced variant, the values used for the parameters defining the influence of the different pools are p_{good}=0.65 and p_{indiff}=0.25. The additional parameters of Section 3 were good_min = 0 and bad_max = 0. For Mutation, we only used the three kinds we discussed in Section 2.2 with the same probability.

The game variants with fixed start positions have these positions depicted in Figure 1. Also in Figure 1, on the right side, we present names for the different agent shapes used in the experiments. In variants 1,8,9,and 11 the goal of the game is to "kill" the prey(s), i.e. a hunter occupying a grid field that is also occupied by the prey (for variant 8, both preys have to be killed at the same time). In all other variants, the game goal is to immobilize the prey. In game variants with only one hunter, this hunter's strategy is to be learned. In variants 2 and 7, both hunters' strategies are learned, resp. both hunters perform on-

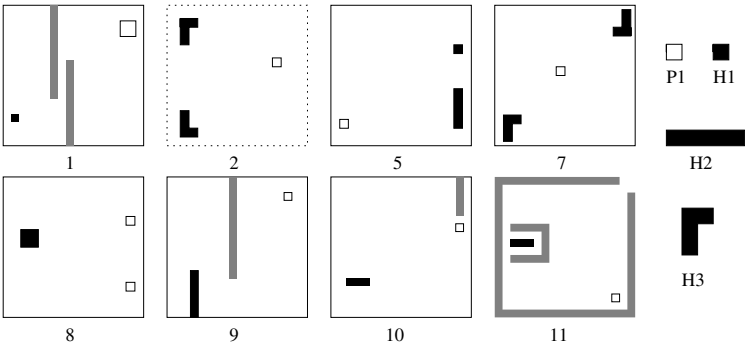


Fig. 1. Start positions for game variants and shapes of agent types

line learning. In variant 3, we have a hunter of type H1 (learning) and one of type H2 with a fixed strategy (simply moving towards the prey, which forces the other hunter to come up with a good support strategy). This fixed strategy is also employed by one hunter of type H2 in variant 4, the hunter of shape H2 in variant 5 and the hunter of type H3 in variant 6. The learning hunter in variant 4 also has shape H2, while the learning hunter in variant 6 has shape H1. The preys in variants 1,3,4,6 and 8 to 11 use a strategy that tries to evade the nearest hunter, while in the other variants the preys move randomly. If not otherwise depicted, the preys are of type P1. Note that variants 2 and 11 have an infinite grid, while all other variants are played on a 30×30 grid.

Let us first look at the results for off-line learning in Table 1. With the exception of the maze variant 1, both the pure success-based and the success-influenced modifications outperform the original algorithm. For many variants, we have large improvements in run (i.e. learning) time and where the improvements are not so big, we see large reductions in the number of steps, i.e. in the quality of the found strategies. Comparing our two modifications, for 8 of the variants the success-influenced approach is faster (often substantially), while for the other two it is not much slower.

For on-line learning (see Table 2), we again have no improvement by either of our modifications for the maze variant 1, while for all other game variants the success-influenced algorithm has better run times, better average number of steps and (with one exception) a higher success rate. The pure success-based method is always better than the base algorithm in at least one measure, but not consistently with all measures. If we compare our two modifications then the success-influenced one has the upper hand for most variants and for those where it does not, it is very close (for variant 9, the run time is not so close, but the average number of steps is better, instead).

So, with the exception of variant 1 (in fact, we tried several other, more complex mazes and variant 1 is typical for the outcome), including accountability of SAPs into EL leads to substantial improvements with regards to both learning time and quality of the found (cooperative) strategies. Why do our modifications

lead to worse performances for mazes? Looking at the difference in definition and performance between the pure success-based approach and the success-influenced approach (that we propose as the approach to choose in future) helps us to find the reasons behind it. And these reasons go back to the general problem of exploration vs. exploitation that learning approaches of behavior have.

Obstacles in general, but very especially mazes require from an agent a lot of exploration in order to deal with them. Especially with a function δ that does not take into account obstacles or other agents that are in the way, it is very important that agents explore their possibilities long enough to get into situations that are clearly better. With a maze, rather long action sequences are required to reach a situation that obviously leads towards the goal (with some intermediate situations that even might look like they lead away from the goal) and therefore allows for an appropriate reward for the decisions that led to the actions. In our base algorithm, the random effects that an evolutionary algorithm makes use of are responsible for exploring the possibilities. Adding accountability to the approach by using the statistic data counters the random effects, obviously more in the pure success-based approach than in the success-influenced one. Taking away some of the randomness makes it harder to explore and consequently our improvement is not an improvement at all but instead makes learning worse. In less extreme situations, with more realistic obstacles, the success-influenced approach on the one side focuses the randomness, which results in being better than the base algorithm. On the other side, by having the accountability of the SAPs just as an influence and not as the only selection criteria on the SAP level, there is the right amount of randomness there to explore the situations, which results in the success-influenced approach being better than the pure success-based approach.

6 Related Work

While using different learning algorithms on different levels of an agent has been suggested as the future of learning in multi-agent systems (see, for example, [9]), our improvement of EL by adding accountability does tackle just one level of learning (although both, our improvement and different algorithms on different levels can be seen as combinations of learning approaches, see the next paragraph).

Within evolutionary computing, learning classifier systems (LCS) are also used to learn the behavior of an agent. In fact, as pointed out in [7], LCS can be seen as a more general technique than reinforcement learning, being able to mimic it. In LCS, the whole set of individuals (that represent single rules) at any point in time represents one agent strategy, so that the fitness of an individual has to be seen as a measure for a strategy component (this is often referred to as the Michigan approach to evolutionary computing). In contrast, our basic EL approach has as individual still a whole strategy (this is called the Pittsburgh approach) and our improvement adds accountability of components on a lower level. Consequently, we still have the advantages of the basic approach, like

not having to completely solve the credit assignment problem and judging a whole strategy after a complete run, but now combined with the accountability advantage of LCS (and reinforcement learning).

7 Conclusion

We have presented an improvement to evolutionary learning of cooperative behavior for agents based on prototypical situation-action pairs and the nearest-neighbor rule. The improvement aims at accountability of all decisions with regard to learning, adding a second layer to learning structure within the basic genetic operators. In our experiments, we used two different ways to make use of this and our evaluation showed that, with the exception of mazes, both improvements achieved better results than the original evolutionary learning algorithm, i.e. the time spent for learning was less or the quality of the learned strategies was better or both. When comparing the two ways of making use of accountability, a success-influenced approach that combines accountability of SAPs with some random influences in most cases achieved better results, due to a better mixture of exploration and exploitation in it, than a pure success-based approach. Since maze-like settings can be easily detected, our results recommend to employ our success-influenced approach whenever the setting is not maze-like.

References

1. M. Benda; V. Jagannathan and R. Dodhiawalla. An Optimal Cooperation of Knowledge Sources, Technical Report BCS-G201e-28, Boeing AI Center, 1985.
2. J. Denzinger and S. Ennis. Being the new guy in an experienced team – enhancing training on the job, Proc. AAMAS-02, ACM Press, 2002, pp. 1246–1253.
3. J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, AAAI Press, 1996, pp. 48–55.
4. J. Denzinger and M. Kordt. Evolutionary On-line Learning of Cooperative Behavior with Situation-Action-Pairs, Proc. ICMAS'00, IEEE Press, 2000, pp. 103–110.
5. T. Haynes, R. Wainwright, S. Sen and D. Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies, Proc. 6th GA, Morgan Kaufmann, 1995, pp. 271–278.
6. J. Hu and M.P. Wellman. Multiagent reinforcement learning: theoretical framework and an algorithm, Proc. 15th Machine Learning, AAAI Press, 1998, pp. 242–250.
7. P.L. Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective, Technical Report 00-03, Politecnico di Milano, 2000.
8. M. Manela and J.A. Campbell. Designing good pursuit problems as testbeds for distributed AI: a novel application of genetic algorithms, Proc. 5th MAAMAW, 1993, pp. 231–252.
9. P. Stone. Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer, MIT Press, 2000.
10. M. Tan. Multi-agent reinforcement learning: Independent vs cooperative agents, Proc. 10th Machine Learning, Morgan Kaufmann, 1993, pp. 330–337.
11. C.J.C.H. Watkins. Learning from Delayed Rewards, PhD thesis, University of Cambridge, 1989.