Combining coaching and learning to create cooperative character behavior

Jörg Denzinger
Department of Computer Science
University of Calgary
denzinge@cpsc.ucalgary.ca

Chris Winder
Department of Computer Science
University of Calgary
winder@cpsc.ucalgary.ca

Abstract- We present a concept for developing cooperative characters (agents) for computer games that combines coaching by a human with evolutionary learning. The basic idea is to use prototypical situation-action pairs and the nearest-neighbor rule as agent architecture and to let the human coach provide key situations and his/her wishes for an associated action for the different characters. This skeleton strategy for characters (and teams) is then fleshed out by the evolutionary learner to produce the desired behavior. Our experimental evaluation with variants of Pursuit Games shows that already a rather small skeleton –that alone is not a complete strategy– can help solve examples that learning alone has big problems with.

1 Introduction

From the first computer games on, there have been games that confront the game player with not just one opponent (i.e. "the computer") but several entities in the game that, according to design, act as opponents (although from the game implementation perspective, there often was only one opponent with many "bodies"). Later, the human player has also become responsible for several "characters", be it the players of a team sports game or a party of adventurers in computer versions of role-playing games. Naturally, under sole human control, the characters do not act together as well as a computer controlled opponent could direct them.

Nowadays, computer games offer a user several fixed character control scripts from which he or she can choose and naturally each character can also be taken over by the user for a more fine-tuned strategy. But both computer controlled opponents and side-kicks for a user-controlled character are far away from acting on a human-like level, in fact, often they are acting less intelligent than a pet and require from a human player a lot of skill in jumping the control between the characters of the team to execute the strategy the player has developed.

In this paper, we present an approach that combines techniques for learning cooperative behavior for agents with the possibility for a user to develop (and tell the agent) some basic ideas for solving a problem. This combination allows us to overcome the basic drawbacks of the two individual ideas, namely the problem that players are not programmers, so that we cannot expect them to write their own character control scripts, and the fact that automated learning of behavior requires providing the learner with a lot of experiences, too many experiences for most human players to "endure". In addition, we see our combined approach also as a good method for game developers to develop more complex, and possibly more human-like, scripts for their non-player characters.

Our approach is based on the evolutionary learning approach for cooperative behavior of agents presented in [DF96], [DE02], [DE03a], and [DE03b]. The basic agent architecture used in these papers are sets of prototypical situation-action pairs and the nearest-neighbor rule for action selection. In any given situation, an agent uses a similarity measure on situations to determine the situation-action pair in its set that is most similar to the situation it faces and it then performs the action indicated by this pair. This resembles the drawing board a coach uses to explain to his/her players game moves he/she wants them to perform. The obvious advantages of such an architecture are that there will always be an action indicated to an agent and that there is quite some resistance to "noise" in the architecture. If an agent's position is a little bit away from a particular position this will in most cases be tolerated by the use of a similarity measure (if being a bit away is not crucial). Additionally, situation-action-pair sets have proven to be learnable by evolutionary algorithms.

The basic idea of our approach is to let the user define basic and key situations and the necessary or wanted actions for all characters involved on his/her side. Then we use the evolutionary learning approach to add to this skeleton of a strategy additional situation-action pairs that are needed to make the user's idea work. Our experimental evaluation shows that rather small and obvious skeletons combined with learning can solve problems that learning alone is incapable to solve (or at least has big problems with). And the used skeletons alone were not able to solve these examples.

2 Basic definitions

The characters of a computer game interact within the game or specific parts of it. As such, the characters can be seen as agents of a multi-agent system with the game being the environment in which these agents are acting. The area of multi-agent systems has established itself over the last two decades and we will be using its terminology to present our ideas. Obviously, the key component of a multi-agent system are agents. There is no agreed-upon definition of what an agent is, since there are a lot of properties of agents that different people are interested in and consequently want them to be covered by their agent definitions.

On a very abstract level, an agent $\mathcal{A}g$ can be described by three sets Sit, Act, Dat and a function $f_{\mathcal{A}g} : Sit \times Dat \rightarrow Act$. The set Sit describes the set of situations $\mathcal{A}g$

can be in (according to $\mathcal{A}g$'s perceptions), Act is the set of actions $\mathcal{A}g$ can perform and Dat is the set of all possible values of $\mathcal{A}g$'s internal data areas. Internal data areas is our term for all variables and data structures representing $\mathcal{A}g$'s knowledge. This knowledge can include $\mathcal{A}g$'s goals, $\mathcal{A}g$'s history and all other kinds of data that $\mathcal{A}g$'s designer wants it to store. The function $f_{\mathcal{A}g}$, $\mathcal{A}g$'s *decision function*, takes a situation and a value of its internal data areas and produces the action that $\mathcal{A}g$ will take under these circumstances. It should be noted that $f_{\mathcal{A}g}$ can be rather primitive, looking up actions in a table based on situations only, for example, but also very complex involving a lot of complex inferences based on the current value from Dat, for example, to form complex plans.

In a multi-agent setting, usually the elements of the three sets from above can be more structured to reflect the fact that several agents share the environment. Each situation description in Sit will usually have a part dealing with other agents and a part dealing with the rest of the environment. The set Act will contain a subset of actions aimed at communicating and coordinating with other agents and a subset of actions not related to other agents (like the agent moving around). If an agent can perform several actions at the same time, often such "combined" actions have a communication part and a part not related to other agents (often there will also be a part manipulating the internal data areas). Finally, due to the unpredictability introduced by having several agents, the knowledge in the internal data areas is often divided into sure knowledge about other agents, assumptions about other agents and, naturally, the knowledge the agent has about itself (with the latter often being structured much more detailed).

The particular agent architecture we are using for our work are prototypical (extended) situation-action pairs (SAPs) together with a similarity measure on situations applied within the Nearest-Neighbor Rule (NNR) that realizes our decision function. More precisely, an element d of the set Dat of our agents consists of a part d_{SAP} and a part $d_{Rest} \in Dat_{Rest}$ (i.e. $d = d_{SAP}d_{Rest}$), where d_{SAP} is a set of pairs (s',a). Here s' denotes an extended situation and $a \in Act$. An extended situation s' consists of a situation $s \in Sit$ and a value $d_{Rest} \in Dat_{Rest}$ (this allows us to bring the current value of Dat into the decision process; note that not a full "state" from Dat_{Rest} is required, often only some parts of a d_{Rest} are used or no part of such an element at all). So, by changing the d_{SAP}-value of an agent, we can influence its future behavior, and even more, the agent can also change its d_{SAP}-value itself.

For determining what action to perform in a situation s with Dat-value $d_{SAP}d_{Rest}$, we need a similarity measure sim that measures the similarity between sd_{Rest} and all the extended situations s'_1,...,s'_m in d_{SAP}. If $sim(sd_{Rest},s'_i)$ is maximal, then a_i will be performed (if several extended situations in d_{SAP} have a maximal similarity, then the one with lowest index is chosen; in contrast to rule-based systems, no other conflict management is necessary). Naturally, it depends on the particular application area how these general concepts are instantiated.

If we have an agent $\mathcal{A}g$ based on a set of (extended) SAPs, then we can define this agent's behavior $\mathcal{B}$ as follows: if $s_0d_{Rest_0}$ is the extended situation from which the agent starts, then $\mathcal{B}(\mathcal{A}g,s_0d_{Rest_0})$ = $s_0d_{Rest_0}$,sap_1,$s_1d_{Rest_1}$,sap_2, ... ,$s_{i-1}d_{Rest_{i-1}}$,sap_i,$s_id_{Rest_i}$,..., where sap_j is an element in $d_{SAP_{j-1}}$, i.e. the set of SAPs that guided the agent at the time it made the decision. Naturally, $s_jd_{Rest_j}$ is the extended situation that is the result of $\mathcal{A}g$ applying the action associated with sap_j and *of the actions of all other agents after s_{j-1} was observed by $\mathcal{A}g$*. The SAPs in the behavior of an agent can be seen as the justifications for its actions. Note that for other agent architectures we might use different justifications, but describing the behavior of an agent by sequences of (extended) situations that are influenced by others is a rather common method.

3 Evolutionary learning with SAPs

As stated in the previous section, SAPs together with NNR provide a good basis for learning agents, since the strategy of an agent can be easily manipulated by deleting and/or adding SAPs. Note that one SAP in a set of SAPs can cover a lot of (extended) situations. We found the use of an evolutionary algorithm, more precisely a Genetic Algorithm for individuals that consist of a set of elements, a very good way to perform learning of a good strategy for a given task (see [DF96]).

The general idea of evolutionary learning of strategies for agents in our case is to start with random strategies, evaluate them by employing them on the task to solve (or a simulation of it) and then to breed the better strategies in order to create even better ones until a strategy is found that performs the given task. It is also possible to continue the evolutionary process to hopefully get even better strategies and so to find the optimal strategy for a task or at least a very good one (since evolutionary algorithms usually cannot guarantee to find the optimal solution to a problem). The crucial points of this general scheme are how a strategy is represented and how the performance of an agent or an agent team is measured. Genetic Operators and their control depend on these two points.

Following [DF96], we have chosen to have an individual of our Genetic Algorithm representing the strategies of all agents in a team that are supposed to be learning agents. More formally, an individual $\mathcal{I}$ has the form $\mathcal{I}$ = ($\{\text{sap}_{11},...,\text{sap}_{1m_1}\}$,...,$\{\text{sap}_{n1},...,\text{sap}_{nm_n}\}$) for a team with n learning agents $\mathcal{A}g_1$,...,$\mathcal{A}g_n$. Then $\{\text{sap}_{j1},...,\text{sap}_{jm_j}\}$ will be used as the d_{SAP}-value of agent $\mathcal{A}g_j$.

This representation of an individual already implies that the performance evaluation has to be on the agent team level. For this evaluation, i.e. the fitness of an individual, we want to measure the success of the team in every step of its application to the task to solve. More precisely, due to the dependence on the particular application task, we need a function $\delta : Sit \rightarrow \mathbb{N}$ that measures how far from success a particular situation is. To define the fitness fit_run of an evaluation run for an individual $\mathcal{I}$, we sum up the δ-value of each situation encountered by the agent team. More precisely, if $\mathcal{A}g_j$ is one of the learning agents in the team

and $\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0}) = s_0d_{Rest_0},\text{sap}_1,...,\text{sap}_k,s_kd_{Rest_k}$ is the behavior of this particular agent in the evaluation run, then

$$fit\_run(\mathcal{I},\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0})) = \sum_{i=1}^{k}\delta(s_i)$$

If we assume that all our agents have the same perception of the situations the team encounters, then just using $\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0})$ for one agent is enough for defining fit_run. If the success of a team contains some requirements on the value of the internal data areas of the agents (i.e. on the value from their Dat_{Rest} sets), then we can extend fit_run to take this into account by extending δ (that then has to look at some kind of "extended" extended situations, i.e. extended situations for all agents) and naturally by looking at the behavior of all agents. This can be of a certain interest in the context of using the evolutionary learning together with the coaching we present in Section 4 to develop characters for computer games, since a developer naturally is interested in the internal data areas of a character/agent and wants to use the values of these areas.

If the task to solve does not involve any indeterminism, that might be due to other agents interfering with the agent team or other random outside influences, then the fit_run-value of a single evaluation run can already be used as the fitness-value for the individual $\mathcal{I}$ representing the team of agents. But if there is some indeterminism involved, then our fitness function fit uses the accumulated fit_run-values of several runs. The number r of runs used to compute fit is usually a parameter of the algorithm. Let $\mathcal{B}_1(\mathcal{A}g_j,s_0d_{Rest_0}),...,\mathcal{B}_r(\mathcal{A}g_j,s_0d_{Rest_0})$ be the r behaviors of $\mathcal{A}g_j$ in those r runs, then

$$fit(\mathcal{I},\mathcal{B}_1(\mathcal{A}g_j,s_0d_{Rest_0}),...,\mathcal{B}_r(\mathcal{A}g_j,s_0d_{Rest_0})) = \sum_{i=1}^{r} fit\_run(\mathcal{I},\mathcal{B}_i(\mathcal{A}g_j,s_0d_{Rest_0}))$$

As Genetic Operators, we use the rather standard variants of Crossover and Mutation for sets. If we look at just one agent $\mathcal{A}g_j$ and two individuals $\mathcal{I}_1$ and $\mathcal{I}_2$ that have as SAP-sets for $\mathcal{A}g_j$ $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$ and $\{\text{sap}^2_{j1},...,\text{sap}^2_{jm_j}\}$, then Crossover picks out of $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\} \cup \{\text{sap}^2_{j1},...,\text{sap}^2_{jm_j}\}$ randomly SAPs (up to a certain given limit) to create a new strategy for $\mathcal{A}g_j$. Mutation either deletes an SAP in $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$, or adds a randomly generated SAP to $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$, or exchanges an element in $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$ by a new randomly generated SAP to create a new strategy for $\mathcal{A}g_j$. We can then have Crossover and Mutation on the level of individuals by either just creating a new strategy for one agent (as described above, with the agent chosen randomly) or by creating new strategies for a selection of agents. In our experiments, just modifying one agent in the case of Mutation and modifying one agent but choosing the strategies for the other agents from both parents (randomly) was already sufficient.

4 Coaching characters

SAPs together with NNR are not only a good basis for learning agents, this agent architecture also is very similar to one human method for coordinating the behavior of several persons, a method we call the "coach's drawing board". In many team sports, the coaches can draw little pictures indicating the players in the team and the opponent players and their particular (spatial) relations to each other (similar to a situation) and also indicate for each player in the team an action or action sequence. This is repeated until a complete behavior, covering the most likely alternatives in detail, is presented. And the coach assumes that the individual players will recognize the situations similar to the current one and will select the best of the drawn situations and the indicated action, similar to the SAPs with NNR architecture we described before. Sometimes, additionally some initial signals are exchanged before a play is initiated, which can be seen as producing an extended situation.

So, many humans seem to be rather familiar with the concept of prototypical situations and associated actions and therefore we think that it is relatively easy for a human being to understand agents that are based on SAPs and NNR. And even more, we think that humans can help in coming up with good SAP sets for agents. Given a good interface that allows to observe solution attempts by the learner (resp. the agents using learned strategies) and to input suggestions for SAPs easily (preferably in a graphical manner), "programming" in SAPs should be much easier than programming behavior scripts, and SAPs with NNR (using an "obvious" similarity measure sim) should be understandable by all kinds of game players (and a good development tool for game designers, too). Even more, since the previous work has shown that the evolutionary learning approach for SAPs can solve non-trivial tasks already without help from humans, it is possible to let the human developer or coach concentrate on only the key ideas for a character or a team (meaning key SAPs) for a particular task and let the evolutionary learning develop such a skeleton strategy into a working strategy for the task. If the skeleton strategy is not enough help, it can be extended after observing what the learning accomplishes and where there are still problems. The latter also allows to correct some behaviors of agents, which can be used to produce flaws or weaknesses in some agents (which might be necessary to allow a human game player to win a game).

More formally, we modify the concepts presented in Section 3 in the following manner. If we have n "coached" learning agents $\mathcal{A}g_1,...,\mathcal{A}g_n$, then the d_{SAP}-value of agent $\mathcal{A}g_j$ consists of a set of "coached" SAPs $\{\text{sap}^{co}_{j1},...,\text{sap}^{co}_{j,m(co)_j}\}$ and a set of learned (or evolved) SAPs $\{\text{sap}^{ev}_{j1},...,\text{sap}^{ev}_{j,m(ev)_j}\}$ that together are used in $\mathcal{A}g_j$'s decision making. If we supply the coached SAPs at the beginning and want to learn good strategies for the n agents without changing the coached SAPs, then the learning method of Section 3 can be used as described there, except that an individual $\mathcal{I}$ now has the form $\mathcal{I} = (\{\text{sap}^{ev}_{11},...,\text{sap}^{ev}_{1,m(ev)_1}\},...,\{\text{sap}^{ev}_{n1},...,\text{sap}^{ev}_{n,m(ev)_n}\})$ and that

the coached SAPs are added to the evolved SAPs of an agent before the evaluation runs are performed. Note that we add the coached SAPs before the evolved ones, so that in case of having a coached SAP and an evolved one of the same (minimal) similarity to the current situation the coached one will be preferred.

If we want to change $\{\text{sap}^{co}_{j1},...,\text{sap}^{co}_{j,m(co)_j}\}$ during learning –for example, we might take a look at the evaluation runs of the best individual found after some number of generations and suggest additional coaching SAPs or remove previously used ones– then there are several ways how this can be incorporated into the learning process. [DK00] presented several possibilities, how strategies evolved under slightly different conditions can be merged and a change of the set of coached SAPs for even only one agent definitely produces slightly changed conditions. The most obvious alternative is to throw away the current population of individuals and essentially restart the whole learning process with the new sets of coached SAPs. But this means that we totally lose the experience accumulated so far by the learner.

Another alternative is to just re-evaluate the current generation of individuals (using the new sets of coached SAPs). There is a good chance that the new coached SAPs will boost the evaluation of some of the individuals (if the "coach" knows what he/she is doing), perhaps even allow an individual to fulfill the given task, which naturally is much better than starting from scratch. There are also combinations possible, where some of the evolved individuals "survive" to be re-evaluated and other members of the new population are randomly generated.

5 Experimental evaluation

We have tested our method for coaching characters combined with evolutionary learning within the OLEMAS system (see [DK00]), a testbed for evaluating concepts for learning of cooperative behavior. OLEMAS uses Pursuit Games as application. In the following, we will first take a closer look at Pursuit Games and the many variants that are covered in literature and then describe OLEMAS and especially how the general concepts and methods from Sections 2 and 3 are instantiated. Finally, we will present some case studies regarding the coaching of characters in this environment.

5.1 Pursuit Games

The term Pursuit Games is used in multi-agent systems to describe games where some predator agents hunt one or several prey agents, a scenario that is not only rather common in Nature, but was also used in games like Pacman. For multi-agent systems, the use of Pursuit Games as testbed was first suggested in [BJD85] and since then it has seen at least as many variants in research literature as there are variants of Pacman out there.

The version presented in [BJD85] had 4 dot-shaped hunters going after one dot-shaped prey on an infinite grid world without any obstacles or other agents, having all agents perform all actions with the same speed and in a turn-based manner. This description already shows off many of the features of Pursuit Games that can be modified, like numbers of hunter and prey agents, possible moves and their speed, having a more or less complex world by using obstacles and agents acting as bystanders, providing agents with shapes and so on. Even the condition to fulfill for winning can be varied: in [BJD85], the goal of the hunters was to immobilize the prey, while the ghosts in Pacman definitely were out to kill by occupying the same square as the prey. A rather extensive description of many features of Pursuit Games and possible feature values can be found in either [DF96] or [DS04].

From a learning point of view, Pursuit Games are of interest because of the many variants that really cry out for automatically developing winning strategies for the hunter agents. Naturally, many variants require cooperation between hunters to win the game. They also provide the possibility to study co-evolution of agents in various scenarios. From the computer game point of view, Pursuit Games provide a rather abstract testbed that nevertheless allows for scenarios that capture the essence of character interaction in many games.

5.2 The OLEMAS System

The OLEMAS system (**O**n-**L**ine **E**volution of **M**ulti-**A**gent **S**ystems; but the "O" can also be interpreted as **O**ff) was primarily developed to provide a testbed to evaluate evolutionary learning of cooperative behavior of agents based on SAPs and NNR. But naturally, other kinds of agents and other kinds of learning can also be integrated to work with the part of OLEMAS that simulates a wide variety of Pursuit Games. In the current version of OLEMAS, we have a variety of hard coded decision functions for agents, with quite a selection of such functions for prey agents and a few primitive functions for hunters and other agents.

At the core of OLEMAS is, as mentioned, a simulator for Pursuit Games. The particular game that is simulated is determined by providing values for the various features of Pursuit Games mentioned in the last subsection. This is usually done using a configuration file. Part of these features include a collection of agents, i.e. hunters, preys and bystander agents. Bystander agents are also used to define obstacles in the world. Each agent is defined by its own configuration file that contains the shape of an agent, its possible actions (with the number of game turns each action needs to be executed) and the decision function used by the agent. In Figure 1, we see a screenshot of the graphical interface for OLEMAS (called XOLEMAS; OLEMAS can also run without this interface which usually speeds up learning runs quite a lot). The window in the upper right corner shows the messages by the simulator. In this screenshot, we see reports on the loaded configuration files of agents.

What happens during a simulation is reported by the two windows on the left of Figure 1. The lower window displays the game. The upper window provides more detailed information on a simulation run, like the current step (or turn) number and the agents involved.

OLEMAS also has a component for learning the behav-

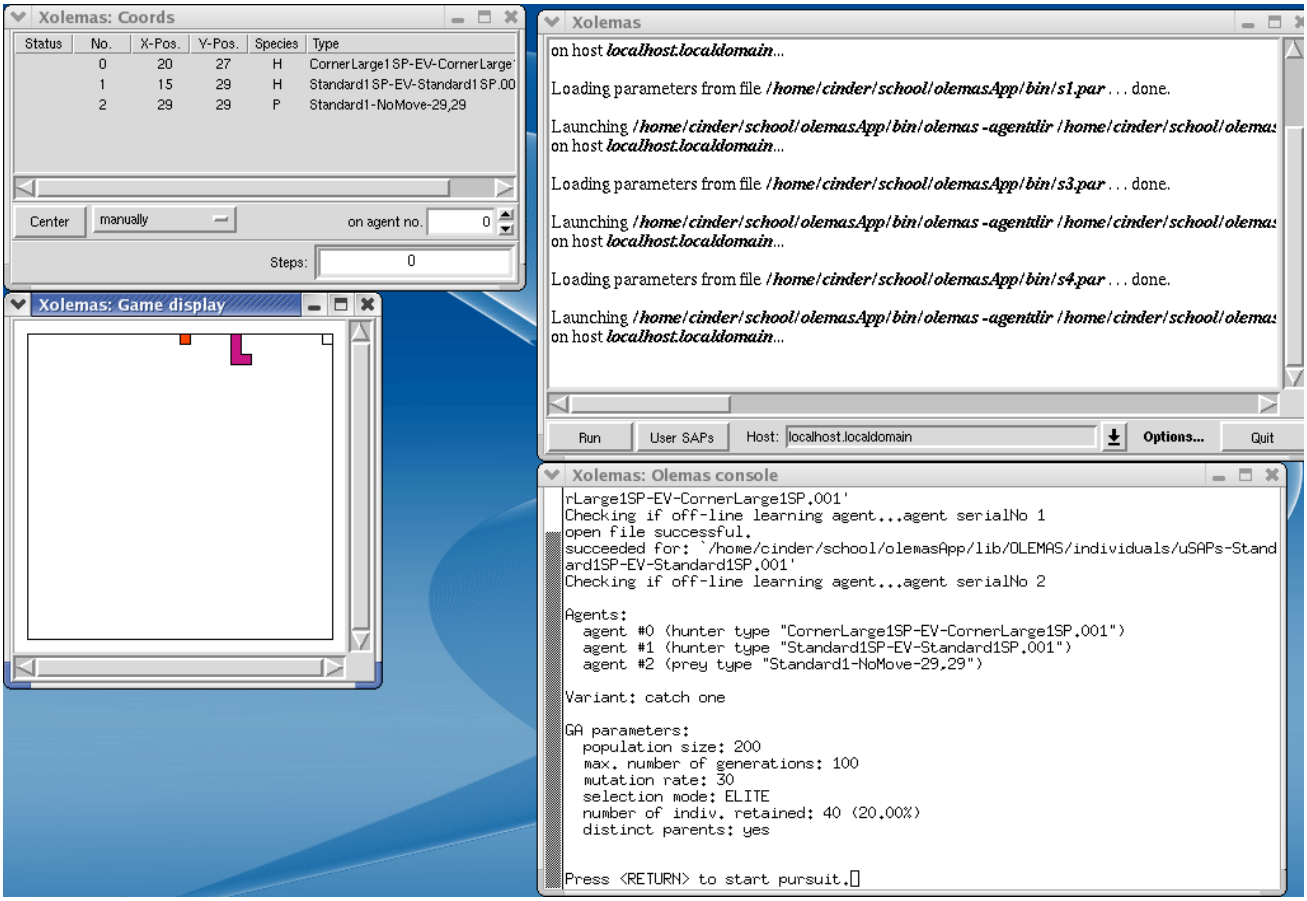

Figure 1: The interface of OLEMAS

ior of agents that implements the evolutionary learning as described in Section 3. The lower window to the right in Figure 1 displays the initial information for a learning run with details about game variant and parameter values for the genetic algorithm. A situation in OLEMAS is a vector that contains for each agent the relative coordinates, the general orientation of it in the world, the agent role (hunter, prey, or obstacle), and the agent type. Relative coordinates mean relative to the agent that is observing the situation. The last line of the text window in Figure 2 presents a situation-action pair (with the starting 0 being the number indicating the action to take).

The similarity measure sim uses only the coordinates and the orientation information and it is defined for two situations s_j and s_k with relative coordinates of agents $\mathcal{A}g_i$ being x_{ij} and y_{ij}, resp. x_{ik} and y_{ik}, and orientation of the agent being o_{ij}, resp. o_{ik} as

$$sim(s_j, s_k) = \sum_{i=1}^{n}((x_{ij} - x_{ik})^2 + (y_{ij} - y_{ik})^2 + ((o_{ij} - o_{ik})^2 \mathrm{mod} 8)).$$

Note that we do not have extended situations in OLEMAS, yet.

For the fitness of an individual, we need the function δ that measures how good a particular situation is. While we have done some work on developing measures that incorporate additional knowledge into this measure (see [DS04]), for this paper we use the rather intuitive idea of measuring how far the hunters are from the prey agents (it is important for coaching to have an intuitive measure). For the distance between two agents we use the Manhattan distance and for δ we sum up the distances between all hunter agents to all prey agents.

For incorporating the idea of coaching, we added a special interface to OLEMAS that is depicted in Figure 2. It allows to enter, resp. delete, the coached SAPs for the individual agents. In the upper left corner, we have the graphical representation of a situation. We can select agents to move them around in this presentation, using the buttons below the situation representation. On the right side, we can select a particular learning agent for which we want to edit the coached SAPs, can look at the coached SAPs, can choose the action that we want to add as SAP and we can remove a coached SAP. Note that we did not put very much work into this interface, an interface for a game developer or a game player would have to be much more sophisticated. This interface can be activated before any learning takes place or at any time the learning performed by OLEMAS is interrupted, which is determined from the main interface. If the user makes changes to the set of coached SAPs, then the current population of individuals is re-evaluated using the new coached SAPs and the learning continues.

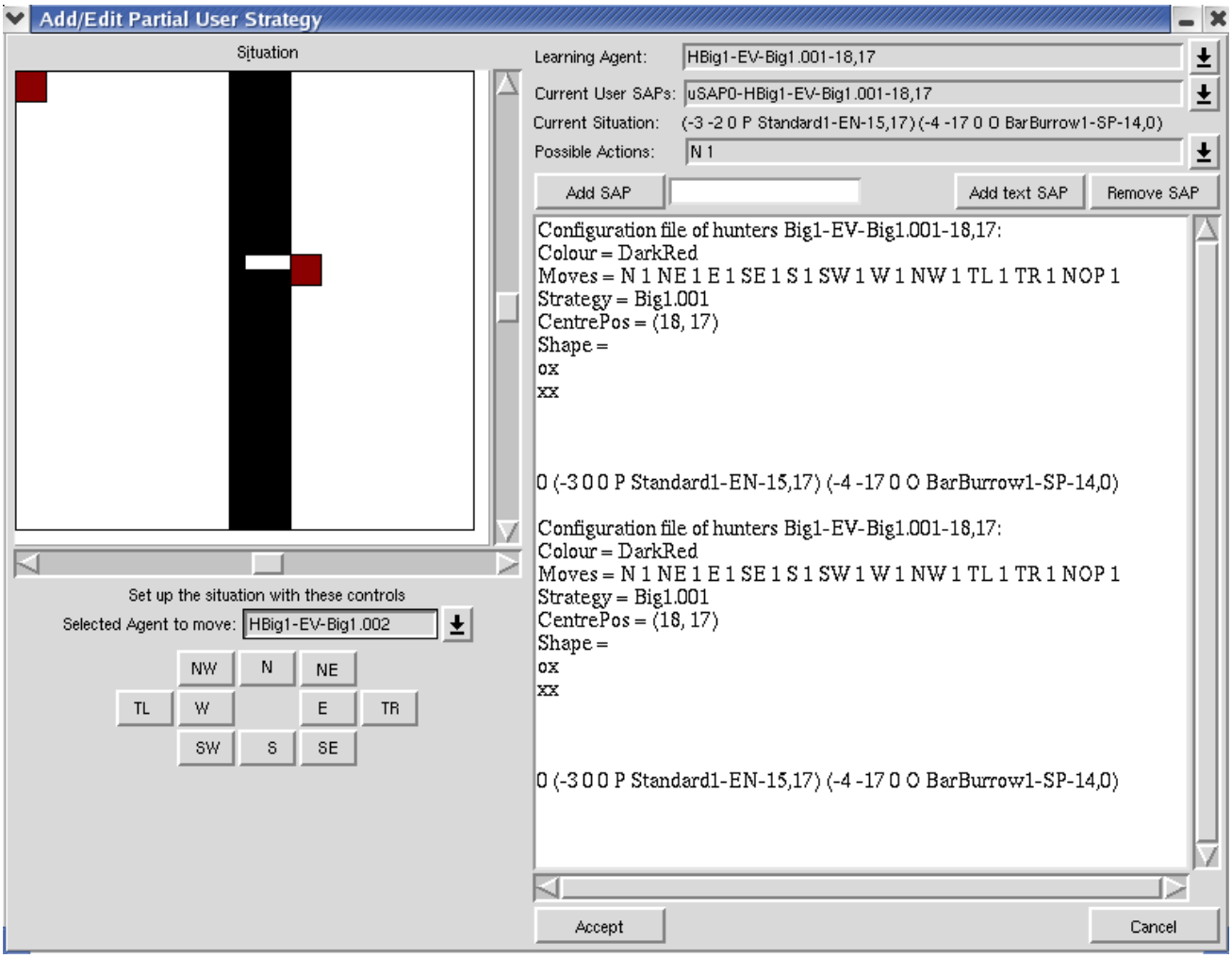

Figure 2: The coaching interface of OLEMAS

5.3 Experiments

We performed several experiments with various variants of Pursuit Games within OLEMAS. Due to lack of space, we can not present all of them and therefore selected 3 examples that show the advantages of combining coaching and learning, but also give insight into problems that the combination has so far. In order to provide some idea regarding the effort necessary for learning, we do not develop the coached SAPs interactively, but report on runs where the coached SAPs are given to OLEMAS at the beginning.

Figure 3 presents the start situations for the 3 examples. In all examples, all agents can move in all 8 directions (but no rotations) and each action takes 1 turn (step) to be completed. The prey agents (light colors) try to stay as far away as possible from the nearest hunter agent (hunters are depicted with dark colors). In all examples, the hunters goal is to “kill” the prey by moving on the same square with it.

In the first example, we have the prey agent hiding within the obstacle. Due to the used fitness measure, once the hunter reaches the obstacle, strategies where the hunter moves to the left of the prey are not favored because the hunter would be moving ”away” from the prey, so the prey does not move. As Table 1 shows, the learner is not able to come up with a successful strategy (within our limit of 100 generations). Figure 4 presents the two coached SAPs that are needed to help the hunter. By sending the hunter left to the prey, the prey is flushed out of its hole and then the learner can take care of the rest. Table 1 shows that the remaining task for the learner is still difficult (one of our 10 runs was not successful within 100 generations), but the coached SAPs make a big difference. The coached SAPs alone are not able to catch the prey.

The second example presents a cooperative variant of the first example. Again, the prey is in a hole and has to be flushed out. The right hunter cannot do this alone, its colleague has to get near to the prey, while the right hunter is away, so that the prey will run away. Here learning alone can be successful (remember, we are using evolutionary learning and the random effects involved produce different learning runs each time), but the 100 generations can be too short. Using the coached SAPs from Figure 5 (the top 4 are for the right hunter, the bottom 2 are sufficient for the left one) together with learning we are always successful in catching the prey. The coached SAPs are far away from a successful strategy, they just achieve the flushing out and the learned SAPs are needed for the catch. The right hunter

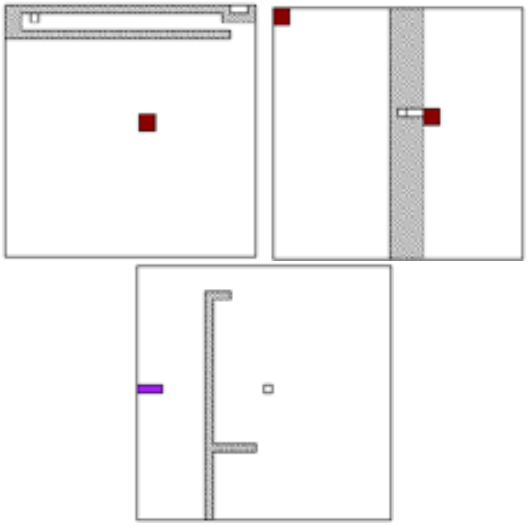

Figure 3: The start situations

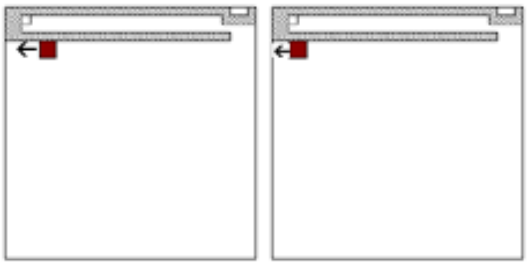

Figure 4: Coached SAPs for first example

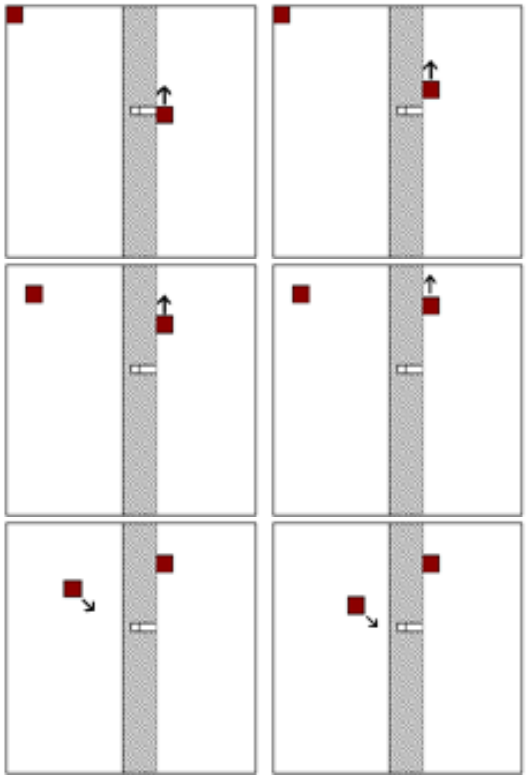

Figure 5: Coached SAPs for second example

also needs learned SAPs to get in a position to do its job.

It should be noted that the average number of steps needed for catching the prey in the successful runs of the learner alone is smaller than the average number for learning together with the coached SAPs. We search only for the first strategy that is successful. The particular coached SAPs we used here are so helpful that many sets of learned SAPs are successful together with them, so that we had a few rather curious strategies as results of the learning runs. In addition, the successful runs where the learner succeeds alone are different from those of the combined approach with the learner and coached SAPs. In these runs, the hunter on the right did not move up at all and the prey simply ran directly into it whereas in the runs with learning and coaching the prey was caught using the desired approach to the problem. This shows that not only can the coached SAPs help to solve the task but can direct the learning in the direction of a specific desired solution.

If we look at the coached SAPs for the hunter on the right in example 2, we can observe that they express the same idea: get away from the prey! Naturally, the question is, why are 4 SAPs needed. For an explanation, look at the third example and the coached SAPs that guaranteed success of the learner (see Figure 6). Navigating large obstacles has been identified as a weakness of the evolutionary learning method we use in [DK00], due to the fitness function that does not consider obstacles in the way. With coached SAPs this problem can be overcome, but several SAPs are needed to reinforce the idea of moving around the obstacle. Runs with less SAPs giving the right direction were not successful and we observed the agents going through a ping-pong effect: the learner often generated SAPs that negated the effects of the coached SAPs. But in interactive mode, this can be easily fixed.

All in all, the combination of learning and coaching is quite successful, allowing the learner to overcome it's weaknesses while still making use of it's strengths. We think that this can be a powerful tool for the development of character behavior. It would allow the human "behavior designer" to simply express the key idea for the behavior, while the learner takes care of making the idea work.

6 Related Work

There are many papers concerned with improving learning by integrating more knowledge. For our work, the following works have some relevance. In [AR02], reinforcement learning was used to learn parameter values of function skeletons, that a user defined to solve a cooperative task. We not only use a different learning method, we use a different agent architecture that does not require programming skills (at least, if combined with learning and if applied to coordination and movement problems).

Improving scripting is an important issue in commercial computer games. But most of the work focuses on making scripting easier by introducing higher level concepts (see [MC+04]). The use of learning techniques, especially evolutionary methods, has been suggested in [ML+04] and

Exp.	Learning only			Coached SAPs only		Learning With Coaching		
	Succ. Rate	Avg. Steps	Avg. Learn Time	Succ. Rate	Avg. Steps	Succ. Rate	Avg. Steps	Avg. Learn Time
1	0%	–	305.7s	0%	–	90%	55.6	73.6s
2	40%	24.8	434.5s	0%	–	100%	61.4	159.4s
3	0%	–	450.9s	0%	–	100%	66.4	54.5s

Table 1: Learning vs. pure coaching vs. learning and coaching

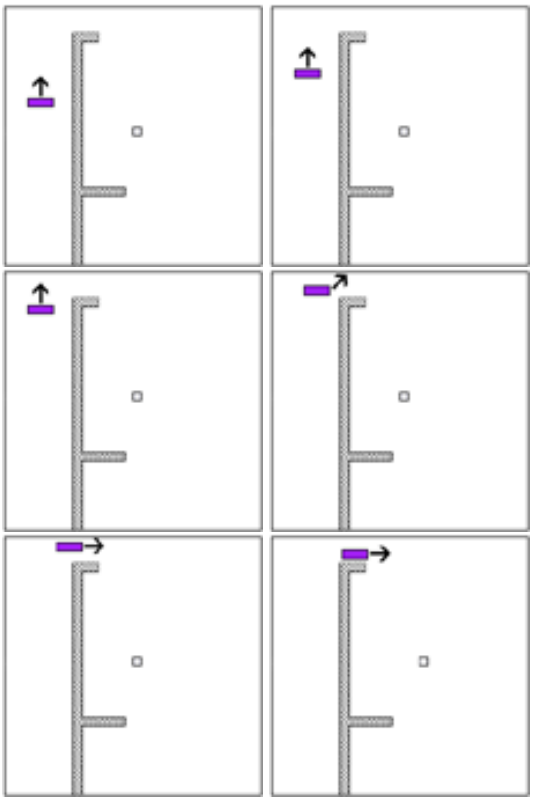

Figure 6: Coached SAPs for third example

[FHJ04] for generating better behavior of non-player characters, but the learning was on the level of parameters for existing functions/scripts, which allows less flexibility for the learner since the level of influence is on a higher level.

7 Conclusion and Future Work

We presented a method that combines learning of cooperative behavior with human coaching for agents that use prototypical SAPs and NNR as agent architecture. Providing the key behavior idea through the use of coached SAPs is very well enhanced by the evolutionary learning approach that "fills in" the other needed SAPs to produce a successful strategy. Our experiments show that the combined approach is very successful for scenarios in which the learner alone has problems. Our method is aimed at helping both game developers that have to create non-player characters and game users that want to create cooperative behavior of their characters that goes beyond the use of built-in scripts without having to write programs.

In the future, we want to address the problem where sometimes the learner tries to undo the effects of the coached SAPs. It seems that more serious changes of the evolutionary learning approach might be needed. We also want to research the effects that different fitness functions and similarity measures for situations have with regard to the ease of coaching.

Acknowledgements

This work was supported by IRIS within the IACCG project.

Bibliography

[AR02] D. Andre and S.J. Russell. State Abstraction for Programmable Reinforcement Learning Agents, Proc. AAAI-02, Edmonton, AAAI Press, 2002, pp. 119–125.

[BJD85] M. Benda, V. Jagannathan and R. Dodhiawalla. An Optimal Cooperation of Knowledge Sources, Technical Report BCS-G201e-28, Boeing AI Center, 1985.

[DE02] J. Denzinger and S. Ennis. Being the new guy in an experienced team - enhancing training on the job, Proc. AAMAS-02, Bologna, ACM Press, 2002, pp. 1246–1253.

[DE03a] J. Denzinger and S. Ennis. Improving Evolutionary Learning of Cooperative Behavior by Including Accountability of Strategy Components, Proc. MATES 2003, Erfurt, Springer LNAI 2831, 2003, pp. 205–216.

[DE03b] J. Denzinger and S. Ennis. Dealing with new guys in experienced teams - the old guys might also have to adapt, Proc. 7th IASTED ASC, Banff, ACTA Press, 2003, pp. 138-143.

[DF96] J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.

[DK00] J. Denzinger and M. Kordt. Evolutionary On-line Learning of Cooperative Behavior with Situation-Action-Pairs, Proc. ICMAS-2000, Boston, IEEE Press, 2000, pp. 103–110.

[DS04] J. Denzinger, and A. Schur. On Customizing Evolutionary Learning of Agent Behavior, Proc. AI 2004, London, Springer, 2004, pp. 146–160.

[FHJ04] D.B. Fogel, T. Hays and D. Johnson. A Platform for Evolving Characters in Competitive Games, Proc. CEC2004, Portland, IEEE Press, 2004, pp. 1420–1426.

[MC+04] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games, Proc. ASE-2004, Linz, 2004, pp. 88–99.

[ML+04] C. Miles, S. Louis, N. Cole and J. McDonnell. Learning to Play Like a Human: Case Injected Genetic Algorithms for Strategic Computer Games, Proc. CEC2004, Portland, IEEE Press, 2004, pp. 1441–1448.