Web Intelligence and Agent Systems: An International Journal 12 (2014) 309–324
DOI 10.3233/WIA-140298
IOS Press

Learning cooperative behavior for the shout-ahead architecture

Sanjeev Paskaradevan, Jörg Denzinger * and Daniel Wehr
Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB, Canada, T2N 1N4
E-mail: sanjeev.com@gmail.com, {denzinge,dkwehr}@ucalgary.ca

Abstract. We present an agent architecture and a hybrid behavior learning method for it that allows the use of communicated intentions of other agents to create agents that are able to cooperate with various configurations of other agents in fulfilling a task. Our shout-ahead architecture is based on two rule sets, one making decisions without communicated intentions and one with these intentions, and reinforcement learning is used to determine in a particular situation which set is responsible for the final decision. Evolutionary learning is used to learn these rules. Our application of this approach to learning behaviors for units in a computer game shows that the use of shout-ahead using only communicated intentions in the second rule set substantially improves the quality of the learned behavior compared to agents not using shout-ahead. Also, allowing for additional conditions in the second rule set can either improve the quality or worsen it, based on what type of conditions are used.

Keywords: Cooperative systems, reinforcement learning, evolutionary learning, communication, shout ahead

1. Introduction

Communication is an important part of many of the basic cooperation concepts for agents: be it selected good results of an agent that boost the problem solving state of another agent (see [10]), be it descriptions of subproblems that allow for bidding by other agents to solve them (see [7]), or be it variable assignments for partial solutions that need to be negotiated over (see [14]) – to name just a few concepts – communication is what allows agents to achieve common goals better and faster than having them try to solve problems on their own. But communication in these cooperation concepts usually comes at a price: the agents need to have a common language based on shared concepts, they have to decide what they communicate when and to whom and what to do with received communications. This requires rather complex reasoning within an agent to get it right and getting it wrong can easily result in behavior of the cooperating agents that is worse than what a single agent would accomplish.

Most probably due to these difficulties around language-based communication, concepts that try to *learn* cooperative behavior of agents have focussed on a much less complex form of communication, namely just looking at what other agents are doing and then adjusting their behavior appropriately (see [15] for an overview). But [9] showed that cooperative behavior can be learned even without observing other members of a team and [24] even reports on examples where communication makes learning more difficult. In contrast, [1] shows advantages out of simple communications for very simple games.

Based on this mixed evidence on the usefulness of communication, we developed a rule-based agent architecture for using communicated information about the intended actions of team members (which we call *shout-ahead*), but also allows agents to reconsider their intended actions to deal with the uncertainties around the environment and the actions of non-team-member agents. Using two sets of rules, one not using any predicates about communicated intentions and one concentrating on communicated intentions but also allowing (selected) other predicates, allows us to make use of rule weights learned on-line via reinforcement learning

*Corresponding author. E-mail: denzinge@cpsc.ucalgary.ca.

to determine if the intent of other agents should be used in the current situation of an agent. Additionally, we use an evolutionary learning approach to evolve these sets of rules within different population pools that reflect different types of agents to be used in a team.[1] For such a team, we assume that it is aimed at solving a particular type of problem, all team members receive the communicated intentions of all other team members, other teams can not listen in to these communications and also cannot manipulate the communications of the agents of the team we learn the behavior for. Since this means that the agents in the team cooperate with each other to solve a common goal, trust of agents is not an issue.

The agent architecture/hybrid learning method combination was motivated by our interest in automatically creating characters for computer games that can be used both as enemies to the characters controlled by the human player but also as support characters for this player. Consequently, we evaluated our new architecture-learning method combination by instantiating it to a computer game, Battle for Wesnoth (see [23]), that allows for both, although in this article we concentrate on fully automated teams of self-controlled cooperating characters to investigate the impact of the shout-ahead communication.

Our experiments with Battle for Wesnoth showed that using shout-ahead substantially improves the quality of the learned behavior (compared to behavior learned using the same resources but without allowing for shout-ahead). We also evaluated different predicate sets around the predicates representing intended actions of the other agents on the team. Our experiments showed that some sets produce worse results than just using intentions but combining intention predicates with predicates about enemy agents produces the best results. Together, the different variants evaluated offer the possibility to create a wide spectrum of opponents for a human player without the need for having a (costly) human developer create the necessary behaviors.

2. Basic definitions and concepts

In this section, we provide a high-level definition of an agent that we will use to structure the presentation of our agent architecture in the next section. After that, we will provide a description of the Sarsa algorithm for reinforcement learning (see [21]), on which we will base the reinforcement part of our hybrid learning method. And we will present the general scheme of evolutionary methods that we will instantiate for the evolutionary part of our learning method.

An agent $\mathcal{A}g$ can be seen as a quadruple $\mathcal{A}g = (Sit, Act, Dat, f_{Ag})$, where Sit is the set of situations $\mathcal{A}g$ can distinguish (its perceptions), Act is the set of actions $\mathcal{A}g$ can perform, Dat is the set of possible value combinations of $\mathcal{A}g$'s internal data areas (essentially the possible internal "states" $\mathcal{A}g$ can have) and $f_{Ag} : Sit \times Dat \rightarrow Act$ is $\mathcal{A}g$'s decision function, taking the current situation and the current value combination of the internal data areas and deciding on the next action to take.

Reinforcement learning is one of the two major concepts for learning of (cooperative) behavior of agents. In our case, by behavior we mean a sequence of actions. As such, the different reinforcement methods try, over time, to create behaviors that are optimal, which is usually expressed by the cummulative rewards an agent receives due to having done a sequence of actions.

Agents that use reinforcement learning usually base their decisions on a matrix within a data area in Dat that contains for each combination of situations and actions an evaluation (*Q-value*) that is updated based on the expected rewards an agent receives while acting in the environment. Usually, the decision function f_{Ag} of an agent $\mathcal{A}g$ selects the action with the highest Q-value (for the given situation) but, in order to achieve some exploration of possibilities, often some random factors are also involved. There are many different ways how the evaluation $Q(s, a)$ of a situation-action pair (s, a) ($s \in Sit$, $a \in Act$) can be updated. We base our method for updating rule evaluation values (see Section 4) on a temporal difference learning method called Sarsa (see [21]).

More precisely, if s is the situation the agent is in at some time t and a is the action taken by it, leading to situation s' (at time $t + 1$) and a reward r and if in situation s' f_{Ag} of the agent selects $a' \in Act$ as action to take, then we update $Q(s, a)$ as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$$

where γ, $0 \leqslant \gamma \leqslant 1$, is the discount rate and α, $0 \leqslant \alpha \leqslant 1$, is the learning factor. Both γ and α need to be chosen by the user of the method, where γ determines the emphasis on the importance of future eval-

[1] While the general shout-ahead architecture and the hybrid learning method were presented in [16], in this article we present additional evaluations, especially focussing on what predicates should be added to the communicated intentions in the second rule set.

uation values and α influences the rate at which the value $Q(s, a)$ converges against the correct evaluation value of the situation-action pair.

In contrast to the step-wise learning by reinforcement learning, evolutionary approaches for learning of cooperative behavior create full behaviors (i.e. complete agents producing a behavior) that are evaluated as a whole. And then the evolutionary process produces new behaviors out of previous behaviors. While there are some evolutionary approaches that evolve the decision function of an agent (for example, via genetic programming), most approaches use an area in Dat, again, that determines the agent behavior and evolve the content of this area (for example, in [9] this area contains a set of situation action pairs and f_{Ag} selects the action of the pair that is most similar to the situation the agent is in).

More precisely, the evolutionary methods work on sets of so-called individuals that they evaluate using a so-called fitness function fit. The next set (called generation) is produced by copying a certain number of best individuals (accoding to fit) from the previous generation into the new generation and filling up the remaining spots with new individuals. A new individual is created by selecting a set of parents (usually one or two) from the previous generation (using a method that tries to include randomness and the fitness of individuals) and applies so-called genetic operators to these parents, resulting in an individual constructed either solely out of "pieces" (genes) from the parents or mostly so, with some (often random) new pieces. After the new generation is created, again all individuals are evaluated using fit and the cycle continues. The first generation is usually created at random and the method stops after a given number of iterations.

3. A rule-based shout-ahead agent architecture

In this section, we present our rule-based agent architecture that allows for making use of communicated intended actions (as a shout-ahead). In the following, we assume that we have a group A of agents in an environment $\mathcal{E}nv$, such that $A = \{\mathcal{A}g, \mathcal{A}g_1, \ldots, \mathcal{A}g_n\}$ with $\mathcal{A}g = (Sit, Act, Dat, f_{Ag})$, $\mathcal{A}g_i = (Sit_i, Act_i, Dat_i, f_{Ag_i})$ for all i, $1 \leqslant i \leqslant n$. In the following, we describe the architecture for $\mathcal{A}g$ in detail.

At the center of our agent architecture is the concept of a *rule with weight*, which has the general form

IF $cond$ THEN a, w.

While obviously $a \in Act$ (Act will always have a do-nothing action as a possible action), the condition $cond$ has to allow for elements from Dat and Sit. In fact, we suggest the concept of a set Obs of observations about a situation and the current value of the data areas of Dat that do not hold rules. While for our architecture two data areas for two rule sets are always part of Dat to allow the evolutionary part of our learner to work on them (see later), there naturally can be other information an agent has that can change and therefore needs to be represented in Dat (usually in its own data area). An example is the current health of a unit in our game application (see Section 5.2).

An element of Obs is a predicate about the environment, other agents or the agent itself and a condition of a rule is simply a set of elements of Obs, i.e. $cond \subseteq Obs$, and a condition $cond = \{obs_i, \ldots, obs_m\}$ is true, if each obs_j is true in the current situation $s \in Sit$ and the current value $d \in Dat$. Formally, evaluating the truth-value of an element is performed by the function $f_{eval} : Obs \times Sit \times Dat \to Boolean$. Naturally, the evaluation of an element of Obs can change, if the situation the agent is in changes or the agent changes its internal knowledge (represented by the value of Dat). Since we are interested in evaluating rule conditions, we extend f_{eval} to $f_{eval} : 2^{Obs} \times Sit \times Dat \to Boolean$ that works on any sets of observations (i.e. all kinds of conditions).

We also distinguish one particular subset Obs_{int} of Obs, where Obs_{int} contains all observations that are communications from other agents in A (about the current situation) indicating what actions these agents intend to take (or have already taken in s). So, Obs_{int} contains all the actions from all the Act_i, but for a given situation s and internal data area value combination d, we have that $f_{eval}(a'_i, s, d) = true$ for at most one $a'_i \in Act_i$, namely the action that $\mathcal{A}g_i$ intends to take in s (if there are communication problems with $\mathcal{A}g_i$ it might not be able to shout ahead its intention and none of the $a'_i \in Act_i$ will be evaluated to true).

As already stated, our agent architecture uses two sets of rules, the set RS and the set RS_{int}, and the current value of Dat stores, among other information, the current instantiations of these sets (in their own data areas). Rules in RS have in their conditions only elements from $Obs \backslash Obs_{int}$ (so, no observations about intended actions allowed), while rules in RS_{int} have only elements from some subset $Obs_{coop} \subseteq Obs$ with $Obs_{int} \subseteq Obs_{coop}$. So, while all observations about intended actions by other agents are in Obs_{coop} (to be used in rules in RS_{int}), we can also allow

for certain other elements of Obs to be used in the rules in RS_{int}. In fact, what elements we allow to be in $Obs_{coop} \backslash Obs_{int}$ has quite an influence on performance, as can be seen in Section 6.3.

The decision function f_{Ag} makes use of these two rule sets in the following manner. In order for communicating an agent's intended action to be useful, f_{Ag} needs to first compute $\mathcal{A}g$'s intended action, communicate it to the other agents, use the communications by these other agents to determine their intentions and then make the final decision what $\mathcal{A}g$'s action will be, followed by actually performing this action. This works obviously well, if the agents take their next action at the same time, but if each agent does this cycle asynchronously from the other agents, then it can happen that the intended action of another agent was already performed and no new communication has reached $\mathcal{A}g$ before it has to make its final decision. While this is not really a problem, if this other agent did exactly the action that it communicated as intended, we have to decide what to do, if communicated intend and really performed action are different. Since the really performed action obviously represents newer information, in the following we will simply use it as what the other agent intended, when evaluating the element from Obs_{int} for this agent. But, as stated above, there will be only one intended action for every other agent that has its corresponding observation evaluated to true. So, reflecting the two rule sets, f_{Ag} essentially consists of two sub-functions, f_{Ag}^{int} and f_{Ag}^{fin}.

For realizing f_{Ag}^{int}, we use the rule set RS in the following manner: First, let RS_C be the set of all rules in RS for which in the current situation and for the given value of Dat their conditions are fulfilled. We subdivide RS_C into the sets $RS_{C_{max}}$ and $RS_{C_{rest}}$ such that for all rules $rl_j, rl_k \in RS_{C_{max}}$ $w_j = w_k$ and for all rules $rl_q \in RS_{C_{rest}}$ we have $w_j > w_q$, i.e. $RS_{C_{max}}$ contains the rules with maximal weight in RS_C. For both sub-functions of f_{Ag} we also need to take into account that our agents are learning, such that it also has to support exploration of possibilities. This is reflected by selecting the rule that determines the action communicated to the other agents probabilistically, but having different probabilities for the different rule groups. More precisely, the probability P of selecting a rule $rl \in RS$ with weight w is defined by

$$P(rl) = \begin{cases} (1-\epsilon) \times \frac{w}{w \times |RS_{C_{max}}|} & \text{if} \quad rl \in RS_{C_{max}} \\ \epsilon \times \frac{w}{\sum_{rl_j \in RS_{C_{rest}}} w_j} & \text{if} \quad rl \in RS_{C_{rest}} \end{cases}$$

The parameter ϵ, $0 \leqslant \epsilon \leqslant 1$, is used to determine the importance of doing exploration (the higher ϵ, the more exploration).

Please note that

$$\sum_{rl \in RS_{C_{max}}} \frac{w_{rl}}{w_{rl} \times |RS_{C_{max}}|} = 1$$

and

$$\sum_{rl \in RS_{C_{rest}}} \frac{w_{rl}}{\sum_{rl' \in RS_{C_{rest}}} w'_{rl}} = 1$$

so that the sum of the probabilities for all rules is 1. In the following, let a^{int} be the action of the rule rl^{int} (with weight w^{int}) that was chosen by f_{Ag}^{int} and that was communicated to all other agents. If RS_C is empty, then a^{int} is the action that does nothing.

f_{Ag}^{fin} uses the same process, i.e. determining which rules in RS_{int} have conditions that evaluate to true, dividing the set of these rules into the set with maximum weight and the rest, determining probabilistically one of these rules using probabilities as defined above and having the action a^{coop} of this rule rl^{coop} (with weight w^{coop}) as the suggested action after taking the communicated intentions of the other agents into account. Again, if no rule in RS_{int} has a condition that is true in this situation, a^{coop} is the action that does nothing.

The final step of deciding on the action that $\mathcal{A}g$ takes is, again, a mixture of using probabilities and looking at the weights of the rules involved. More precisely, if $w^{coop} < w^{int}$, then $\mathcal{A}g$ performs a^{coop} with probability p_{coop}, and with probability $(1 - p_{coop})$ it performs a^{int}. If $w^{int} \leqslant w^{coop}$, then $\mathcal{A}g$ performs a^{coop}. Here p_{coop} is another parameter that can be used to determine how much the agent relies on the shout ahead by other agents.

And, again, if in any situation there are no rules in any of the rule sets in Dat with conditions that evaluate to true, then the agent takes the action that does nothing.

4. A hybrid learning method for cooperative behavior

In this section, we present our hybrid learning method for cooperative behavior for agents using the agent architecture described in the last section. Figure 1 presents the general flow chart for our learning method. As before, we assume that we have a set A of

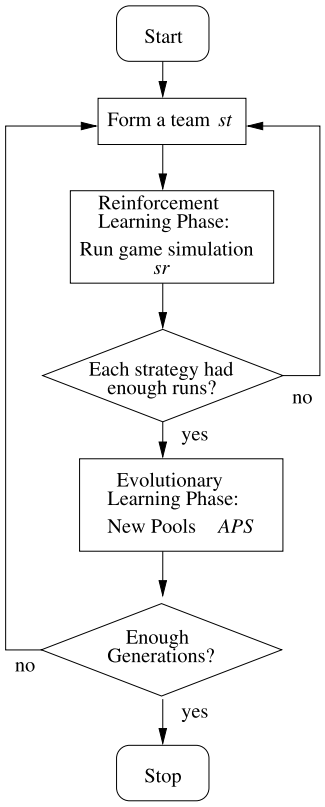

Fig. 1. Learning work flow.

agents and that we are interested in learning the behavior for agent $\mathcal{A}g$. This behavior of $\mathcal{A}g$ is supposed to be very general, allowing the agent to cooperate with different agents (in different teams) and doing this in different scenarios. As a consequence, we analyze the scenarios for all the different roles agents play in them and for each role we create a pool of possible agent instantiations. The pool for $\mathcal{A}g$ represents the population of agent strategies for $\mathcal{A}g$ for the evolutionary part of our hybrid learning method. Naturally, also the other pools can learn, but our method also allows for not-learning pools. For evaluating the different individuals in a pool, we apply each of them in several simulations (using different agents from the other pools for the other roles in the scenario of a simulation). And during each simulation run, all learning agents apply reinforcement learning of the weights of their rules.

More precisely, let $APS = (AP, AP_1, \ldots, AP_k)$ be a group of agent pools such that AP contains possible strategies (individuals) for $\mathcal{A}g$ and for each $\mathcal{A}g_i \in A$, there is an agent pool AP_j such that AP_j contains possible strategies for $\mathcal{A}g_i$. For a scenario $Scen$, we have as *team requirement* TR a certain number of agents from some agent pools, i.e. $TR = (AP'_1, \ldots, AP'_l)$, with $AP'_j \in \{AP, AP_1, \ldots, AP_k\}$. A simulation team $st = (ag_1, \ldots, ag_l)$ needs to fulfill the team requirements TR, i.e. $ag_j \in AP'_j$.

A simulation team is used in a simulation run sr of the scenario $Scen$ (which is the inner loop of Fig. 1). The result of sr is on the one hand side a run fitness result $rfit(sr, ag_j)$ for each of the agents of st and also an update of the weights of the rules for each ag_j that is based on the agent architecture from the last section. This update of the weights is the result of performing reinforcement learning during the run using a modification of the Sarsa method from Section 2. For each action performed in the simulation run, we update the weight of the rule responsible for it in the following way. If rl is the rule and w its weight and if the reward is r and w' is the weight of the rule responsible for the action following the action from rl, then w is updated by

$$w \leftarrow w + \alpha\big[r + \gamma w' - w\big]$$

with α and γ as described before.

It should be noted that applying the Sarsa method to rule weights naturally comes with some problems. While for situation-action pairs a very fine-grained learning can be achieved and convergence against optimal strategies can be proven, rules can be applied in many situations, and for every rule there usually are situations in which its application is not rewarding, although the rule is applicable in the situation. The rule weights aggregate over both good and bad situations and a rule that is often good will have a high weight.

So, while rules are a more compact representation than a matrix of situations and actions, there are many possible sets of rules and we need to find the right two sets for our agent $\mathcal{A}g$. This is the task of the evolutionary part of our hybrid learning method (which is represented by the outer loop in Fig. 1). As already stated, AP contains possible strategies for $\mathcal{A}g$, where a single strategy consists of a set RS and a set RS_{int} of rules. To create the next generation of strategies, we use the usual operators crossover and mutation, although with a slight twist due to the fact that we have available rule weights based on the reinforcement learning process (this was inspired by [8]).

Given two strategies (RS, RS_{int}) and (RS', RS'_{int}), a crossover creates the strategy $(RS^{new}, RS^{new}_{int})$ by selecting the $|RS|$ best rules (with regard to their weights) from $RS \cup RS'$ to create RS^{new} and the $|RS_{int}|$ best rules from $RS_{int} \cup RS'_{int}$ to create RS^{new}_{int}. If RS^{new} or RS^{new}_{int} contain copies of the

same rule with different weights (which becomes more and more likely during the evolutionary process), then the copy with the lower weight is mutated and the weight of this mutated rule is set to 0. In general, the mutation of a rule IF $cond$ THEN a results in changing $cond$ to a $cond^{new}$ with a probability p^{cond} and a to an a^{new} with probability p^{act}. a^{new} is a random action from the set Act of $\mathcal{A}g$, while $cond^{new}$ is created by deleting a random set of observations from $cond$ and adding another randomly chosen set of observations. Naturally, for a rule for RS the added observations are chosen from $Obs \backslash Obs_{int}$, whereas for a rule for RS_{int}, we choose from Obs_{coop}.

Mutation is not only applied within a crossover, it is also a stand-alone operator, creating a new strategy by randomly choosing a rule in either RS or RS_{int} and mutating it as described above. Also in this case the weight for the new rule is set to 0. The first generation of AP is created by creating random rules for RS and RS_{int} for each strategy (naturally fulfilling the requirement that conditions of a rule in RS do not contain any communicated intend observations and that conditions of a rule in RS_{int} only contain elements from Obs_{coop}).

The fitness fit of a strategy (RS, RS_{int}) is created out of applying the strategy in several simulation runs $sr_1, \ldots sr_{s_{max}}$ with different simulation teams, combining the $rfit(sr_i, ag)$-values. Naturally, (some of) the received rewards by the rules will be part of the value computed by $rfit$, as will be the overall result of a simulation run, but the concrete implementation of $rfit$ and fit depends on the application area and we will provide an example in the next section.

As already mentioned, Fig. 1 shows the complete work flow for our hybrid learning approach for the shout-ahead architecture.

5. Battle for Wesnoth

In this section, we provide an instantiation of our hybrid learning method and the shout-ahead architecture for the application of creating good cooperative game characters for the game Battle for Wesnoth (see [23]). We will first give a brief introduction of the game and then present the instantiation of our approach.

5.1. Battle for Wesnoth: The game

Battle for Wesnoth is a turn based strategy game where the player controls an army of units. While the game also includes an economic component that allows for the creation of new units during a game, we did not use this aspect of the game for this work. Instead, we looked at the single player campaign mode where the player battles an AI player (that centrally controls its army of units) and concentrated purely on learning good autonomous controls for units (by having our learning approach evaluating teams of units against the AI player). The intention for this is to create a large variety of unit behaviors that can be put together in different teams to have a large variety of opponents for human players (in contrast to the current AI).

Battle for Wesnoth is played on a hexagonal tile based map. Each hex(agonal) tile is a composite that can consist of terrain types such as rivers, forests, and mountains or built up areas such as bridges, castles, and villages. Each tile imposes a movement cost on the units that enter them. This movement cost is based both on the type of terrain and the type of unit. Each tile also provides offensive and defensive bonuses for each unit, also based on the type of unit, for example Elves have a defensive bonus for being in forrest tiles. There is also a day/night cycle that provides advantages and disadvantages to different units. And tiles with a village on it provide a unit that rests on it healing.

Each player in a Battle for Wesnoth game controls an army consisting of one leader unit and many follower units. All units can either move or attack an enemy unit. Each unit has an allotted number of movement points which it uses to move across the tiles. Because different tiles have different movement costs the distance that a unit can travel often varies. If a unit attacks another unit it forfeits all of its movement points and cannot take any further actions that turn, therefore if a unit needs to be moved to a different tile it must perform the move action prior to performing an attack action. A unit also has the option to not take any action and stay on its current tile. A move action from one tile to another is not guaranteed to succeed because the unit can get ambushed by stealthy enemy units in which case the unit is stopped at the ambush tile. An attack action is not guaranteed to succeed because the success rate is determined by the type of weapon used, the tile the enemy unit is on, the tile the attacking unit is on, and the day/night cycle, all of which combine to form an attack success percentage ranging from 40% to 80%. This makes Battle for Wesnoth non-deterministic in that actions taken by a unit have

Fig. 2. Battle for Wesnoth.

a chance to fail and may not change the game state at all.

A game of Battle for Wesnoth consists of a sequence of turns. Each turn, every player gets their own turn to perform all of the actions for their side. The next side does not get to perform its actions until the previous side has finished. A player wins the game if the leader unit of the other player has died. Figure 2 shows a screen shot of the game with the various units, terrain, and buildings.

It should be noted that the AI player (we have been using version 1.7.12 of the game) is rather strong and even for good human players winning is not at all guaranteed. Therefore having more variance in opponents available definitely improves the gaming experience for human players.

5.2. *Instantiating our approach*

In order to instantiate our approach to Battle for Wesnoth, we first have to instantiate our shout ahead agent architecture, which means we have to provide Obs, Obs_{int}, Obs_{coop} and Act. A unit in Battle for Wesnoth can make observations about all the hex tiles it can move on or can attack. For each of these observable tiles, we created various predicates that describe features of the tile or adjacent tiles, respectively of enemy units on the tile and on adjacent tiles. The set Obs contains all these predicates applied to each observable tile.

To study the influence that the choice of Obs_{coop} has on the success of our learning method, we divided the predicates mentioned above into 3 different sets: Obs_{own}, Obs_{team} and Obs_{enemy}. Obs_{own} contains all observations of the self, which include information about the agent's own hitpoints in the form of predicates being true if the unit has full hitpoints (HasFullHP), less than 75 percent (HasLt75PHP), more than half (HasMoreThanHalfHP) and less than a quarter (HasLessThanQuarterHP) of the full hitpoints.

Obs_{team} contains all observations about the other units of the team of a unit. This includes the observation that a particular tile is adjacent to a friendly unit (AdjacentToFriendly) or adjacent to the team leader (AdjToFriendlyLeader), but also that a tile is adjacent to a damaged friendly unit (AdjacentFriendlyIsHurt) or a healer unit in the team (AdjToHealer). Since a unit cannot move to or attack a tile with a friendly unit on it, we do not need observations about friendly units on a particular tile.

The set Obs_{enemy} contains all observations about the enemy units. This includes observations about units on the tile itself, like the enemy leader being on it (HasEnLeader), an enemy unit on it is of a type that can not retaliate (EnCan'tRetaliate) or that is disadvantaged on the tile (EnIsDisadvantaged). Naturally, there are also observations regarding the remaining hit points of such a unit, namely it having less than half of its full hit points (EnemyHasLessThanHalfHP) and less than a quarter of its full hit points (EnemyHasLessThanQuarterHP). Due to the importance of the leaders, we also have a predicate that evaluates if the unit on a tile is a threat to a unit's leader (EnThreatensLeader) and a predicate that indicates that a tile is closest to the enemy leader (among all the tiles that a unit can reach in a round) (ReachableLocIsClosestToEnLeader). And, similar to the observations in Obs_{team} that look at the neighboring tiles of a tile, we have observations that indicate that there is an enemy unit on a tile adjacent to the tile that is the argument of an observation (AdjacentToEnemy), that the enemy leader is on an adjacent tile (AdjToEnemyLeader), that there is an enemy on a neighboring tile that is disadvantaged on that tile (AdjEnIsDisadvantaged) and that there is an enemy adjacent that can not retaliate (AdjEnCan'tRetaliate). There are also observations regarding the hit points of an adjacent enemy, again indicating that such an enemy has less than half of the full hit points (AdjEnHasLT1/2HP) and less than a quarter of the full hit points (AdjEnHasLT1/4HP). And finally, the importance of a threat to the own leader is looked after by having an observation that an enemy on an adjacent tile threatens the own leader (AdjEnThreatensLeader).

In addition to the above "standard" observations in Obs, we have for Obs_{int} for each observable tile communicated intentions about another unit in the team attacking it (CPAttackingLocation), moving to a tile adjacent to it (CPMovingToAdjLoc) or attacking a tile adjacent to it (CPAttackingAdjLoc). As defined earlier, Obs_{int} is always part of the set Obs_{coop} used in the conditions of rules in the set RS_{int}. In our experiments, we will first evaluate our approach by having $Obs_{int} = Obs_{coop}$ and then extend Obs_{coop} by adding either Obs_{own}, Obs_{team}, or Obs_{enemy}.

For the set Act, we have two parameterized actions, *move(tile)* and *attack(tile)*, where the tile for the move action can only be an empty tile within the range of the remaining move points of the unit and the attack tile can only be a tile adjacent to the current tile (with an enemy unit on it). There is also a pass action that is automatically performed if no rule is triggered. The value of p_{coop} in our experiments was 0.5. For the value of ϵ we chose in our experiments 0.65, i.e. 35% of the time the agent exploits one of the rules with highest weight, while 65% of the time it does exploration.

Since the different units of a player move one by one, we are in the situation mentioned in the last section that some units already moved after they communicated their intentions. As stated, we then assume that the move is the communicated intention. Since a player can decide the order in which units move and agents might make several moves, we had to come up with a way to determine the order in which units move. This is achieved by having a list with all agents that still can move. The system then goes through this list by selecting an agent, letting it decide on an action, performing this action and putting the agent back at the end of the list if there are moves for it left. If none of the agents can perform an action anymore, the turn of the team is over.

For learning, we used 4 agent pools, one for the leader, one for melee type units, one for ranged attack type units and one for healer type units. For the reinforcement learning part, we calculated the reward a strategy for an agent received in the following manner. The rewards for *move* and *attack* actions are calculated differently. For a *move* action, three different rewards can be achieved by performing it and the values for these rewards are added up to generate the complete reward r. The first reward, $ctelr$, is awarded, if the move resulted in getting the unit closer to the enemy leader. The second reward, ghr is given, if the unit ended up on a tile that provides healing (i.e. a village or adjacent to a healer unit). Finally, the third reward, hr, is only given to healer units and only if the move resulted in being adjacent to a friendly unit that is hurt. In our experiments in the next section, we used $ctelr = 125$, $ghr = 60$ and $hr = 350$.

The reward for an *attack* action also combines several individual rewards that, natually, focus on the damage to enemies and the health of the friendly leader. More precisely, we use the following formula:

$$r = (pelhp - elhp) * w_{elhm} + (pehp - ehp) * w_{ehm} + flhp * w_{flhm} + nke * w_{kr}$$

where $pelhp$ is the hitpoint percentage (compared to the beginning of the game) of the enemy leader before the attack action and $elhp$ is the percentage after the attack. $pehp$ is the total enemy hitpoint percentage (again, relative to the start) before the action

and ehp after it. $flhp$ is the current hitpoint percentage of the friendly leader and nke is the number of killed enemies after the *attack* action. w_{elhm}, w_{ehm}, w_{flhm} and w_{kr} are weights for the different rewards and were set as $w_{elhm} = w_{flhm} = 150$, $w_{ehm} = 1200$ and $w_{kr} = 250$. The other parameters around reinforcement learning were set to $\alpha = 0.5$ and $\gamma = 0.75$.

In the evolutionary part of the learning, we learned good strategies for all 4 pools. The parent selection of the operators uses fitness proportionate selection. The fitness computation is, as already mentioned, based on evaluating a strategy from a pool in several simulation runs. In our experiments, the number of simulation runs guaranteed for a strategy was 10 and the teams were put together randomly. For a strategy ag and a simulation run sr, we computed $rfit$ as follows:

$$rfit(sr, ag) = (1-ehp)*w_{ehfm}+flhp*w_{flhm}+wb$$

where ehp, $flhp$ and w_{flhm} are as in the reinforcement rewards measured at the end of the simulation run, w_{ehfm} is a parameter (set to 1000 in our experiments) and wb is a bonus, which, in our experiments, was 2100 if the team of learning agents won and -2100 if the team lost. As alread stated, the fitness value fit for ag is the sum of the $rfit$-values of the 10 simulation runs. The mutation probabilities p^{cond} and p^{act} were 0.2 in our experiments.

There are many possibilities to integrate additional knowledge into the reward and fitness computations. But it was not our goal to create near invincible agents, we wanted to create agents that are not exactly stupid but that also can make mistakes, offering the human player some challenge but still allowing for wins. And, naturally, in this paper we are mostly interested in the potential of the shout-ahead, so that the performance of the hybrid learner without using shout-ahead should leave room for improvement.

6. Experimental evaluation

In this section, we present our experimental evaluation of the shout-ahead architecture/hybrid learning approach from Sections 3 and 4, instantiated to Battle for Wesnoth as described in the last section. We will first present the general set-up of the experiments, then compare our approach using two sets of rules with only using the first set (i.e. without using shout-ahead) and after that look into varying the set Obs_{coop} by adding the different observation sets defined in the previous section to the core set Obs_{int} for usage in the second rule set.

Table 1
Comparison between performance of agents without and with shout-ahead

Exp.	Without shout-ahead		With shout-ahead	
	Win/Loss/Tie	Gen.	Win/Loss/Tie	Gen.
1	3/97/0	52	25/74/1	49
2	5/95/0	58	26/72/2	39
3	7/91/2	50	26/71/3	60
4	10/79/11	48	28/61/11	57
5	11/81/8	57	28/61/11	21
6	12/87/1	25	29/69/2	51
7	13/84/3	33	29/66/5	42
8	15/82/3	60	32/67/1	33
9	17/80/3	54	36/56/8	48
10	28/66/6	52	40/47/13	59
Av.	12.1/84.2/3.7	48.9	29.9/64.4/5.7	45.9

6.1. Set-up

Since there are many kinds of random influences, like the success of an attack in the game itself or in the control of the evolutionary learning component, we need to perform several experiments to be able to make any statements with respect to success or failure of the shout-ahead idea. Also, due to the indeterminism in the game, evaluating a particular team of learned agents cannot be done performing just one game against an opponent, which in our experiments is the AI player that is also used for the learning. Therefore we evaluate each learned team in 100 games against this opponent and compare the achieved wins, losses and ties (indicated in all tables by the Win/Loss/Tie column). A tie is achieved if none of the two players kills the leader of the other player within the given number of game rounds (which is 40 in our experiments). All experiments were performed on an iMac with a 3.06 Ghz Intel Core 2 Duo processor running OSX 10.9.

The scenario of Battle for Wesnoth we used in our experiments is called "The Freelands" and consists of a map with the castle of one player up in the north and the other down in the south. In addition to the leader unit, each player has two melee units, two ranged attack units and two healer units. The result team of a learning run consists of the two best strategies of the pools for these 3 unit types being selected (as is done in the simulation runs during the learning) and all these strategies come from one generation of the evolution-

ary learning component (the best we observed during the 60 generations we have in a learning run, which is indicated in all tables in the Gen.-column). Each agent pool contains 20 strategies and a strategy of an agent had 60 rules (50 in RS and 10 in RS_{int}). Each rule in RS was allowed to have up to 10 predicates, while there were only a maximum of 4 in the rules in RS_{int} (Obs_{int} is small compared to Obs so that this is a good number when using only Obs_{int} as Obs_{coop} and we wanted the same set-up also for the experiments where Obs_{coop} contains more than just Obs_{int}). A learning run takes between 20 and 25 hours, which sounds like a lot but for this application is absolutely acceptable.

6.2. Results without and with shout-ahead

In our first set of experiments, we wanted to evaluate the basic potential of shout-ahead versus just using rules without communicated intentions. This means that for these experiments we used as Obs_{coop} the absolute minimum, which is just Obs_{int}.

As Table 1 shows, on average the agents using shout-ahead are 2.5 times more successful than the agents not using shout ahead (as indicated by the first numbers in the Win/Loss/Tie columns of the table) and the number of ties, again on average, is also higher (as indicated by the third number in the Win/Loss/Tie columns ofthe table). In fact, with the exception of one very successful learning run without shout-ahead, the worst learning run with shout ahead is already quite a bit better than the (second-) best learning run without shout-ahead (we ordered both columns by success) And the best runs for both variants also have the best run with shout-ahead with 12 more wins and 7 more ties.

The causes of this increase in wins can be seen in the rules that the best learning runs created. Because of the small number of observations in Obs_{int} there is a much smaller number of possible rules that can exist in RS_{int} compared to RS, and so it is common for the agents with shout-ahead to create similar rules in RS_{int}, making it easier for agents to evolve complementary rules. This is best illustrated by the following rule, which is usually the top weighted rule in RS_{int} for nearly all agents of the best performing teams in Table 1.

IF $CPAttackingLoc(x) \wedge$
$CPAttackingAdjLoc(x)$
THEN $Attack(x)$

Table 2
Average performance of randomly selected teams without and with shout-ahead

Without shout-ahead	With shout-ahead
Win/Loss/Tie	Win/Loss/Tie
11.4/82.8/5.8	16.6/76.2/7.2

This rule[2] states that the agent should attack the target location if both the target location and a location adjacent to it are the targets of communicated attack intentions. As the other agents in these best teams will regularly also use this rule there is a high chance of attacks being coordinated to kill enemy units faster. As a unit can only attack once per turn, a single friendly unit can not be responsible for both attack observations required by the rule, and so these coordinated attacks must also be between 3 or more units (the two observed, and the agent executing the rule). This tendency to attack in groups appears to be a large part of the success of these teams.

There is a wide variety in the number of the generation of the evolutionary learning in which the best team was found, but overall the amount of computation (measured in generations) is the same for both variants, so that the substantial difference in quality of the agents is due to the shout-ahead architecture.

While this paper is about the shout-ahead architecture, the larger context of our research is around creating behaviors for non-player characters and providing a large variety of such behaviors of different quality automatically, without the large cost that is created by having human developers do this. With this regard, agents using shout-ahead and agents not using it already provide a wide spectrum of quality (as seen in Table 1), especially if several learning runs are used. But even within a single run we do have already an interesting variety of agent behaviors. In order to show this, we used not the best strategies from the learning runs for each of the agent architecture variants but a randomly selected team (out of the pools that provided the best team in Table 1) and again played 100 games against the game AI.

Table 2 shows the average results over the learning runs for this experiment. For both architecture variants, we get teams that are winning (and naturally that are loosing), which means that the teams can be used against humans and provide a lot of variety for op-

[2]We omit the weights for rules in the following, since the weights naturally vary over time due to the contineous reinforcement learning.

Table 3
Performance of agents with Obs_{own} added to Obs_{coop}

Exp.	Best Gen.	Win/Loss/Tie
1	36	0/98/2
2	40	0/98/2
3	58	0/98/2
4	60	0/95/5
5	23	1/92/7
6	26	1/84/15
7	42	5/87/8
8	41	6/94/0
9	49	10/80/10
10	43	15/71/14
Av.	41.8	3.8/89.7/6.5

ponents, even if we do only one learning run. It also should be noted that the average from this experiment for the variant with shout-ahead is still better than the average of the best teams without shout-ahead, again showing the substantial improvement that shout-ahead provides.

6.3. Varying Obs_{coop}

As stated above, so far we used $Obs_{coop} = Obs_{int}$ so that all rules in RS_{int} would only contain observations for communicated intentions. This meant that a decision to act based off of Obs_{coop} had to be ignorant of the state of the game outside of the intentions of other agents, and through its use of intent only it showed how beneficial that communication can be for creating successful agents. In the following experiments we explore the effects of expanding RS_{int} by allowing rules in RS_{int} to contain both communicated intentions and game state information through the inclusion of Obs_{own}, Obs_{team}, or Obs_{enemy}.

6.3.1. Evaluating inclusion of Obs_{self}

In Table 3 we see the results of combining an agent's observations of itself with the communicated intentions of other agents for Obs_{coop}. Many experiments failed to create a best team that could even win a single game out of a hundred. This extreme drop in agent performance was traced to several causes. From an examination of the rules these agents have created in comparison to those of the other experiment sets, it appears that self focused observations work poorly when they are combined with only communicated intentions and no other type of observations, and the agents start to favour self observations and exclude communication. This greatly increases the chance of having multiple self observations in the same rule, which can potentially contradict each other and create a rule that can never be used. This is well illustrated by the following rule, from the most fit melee agent from experiment 6 of Table 3.

$$\begin{aligned}&\textbf{IF } HasLt75PHP(x) \wedge \\ &\quad HasFullHP(x) \wedge \\ &\quad CPMovingToAdjLoc(x) \wedge \\ &\quad HasMoreThanHalfHP(x) \\ &\quad \textbf{THEN } Move(x)\end{aligned}$$

This rule states that the agent should move if it has less than 75% hitpoints, full hitpoints, another agent has communicated that it will move to an adjacent tile and finally that the agent has more than 50% hitpoints. As the first two conditions can never both be true, this rule is non-functional. Every agent has a limited number of rules that can exist in RS_{int}, and so the presence of these non-functional rules has a strong negative effect on an agent's ability to evolve more effective rules. Rules that include the full hitpoints condition also suffer from another issue, due to how attacks are carried out in Battle for Wesnoth. In the game, when a unit attacks, the target is able to counter-attack and deal damage as well. The result of this is that a rule that requires full hitpoints and has an attack action will usually only be used once in a game. While a friendly healer unit may restore some of the lost hitpoints, it is very rare that a unit will be healed multiple times before its next attack to have full hitpoints again.

It is also very common to see agents with rules that lack communication entirely, even as their highest weighted rules, disrupting the benefits we were seeing from communication in our previous experiments. An exception to this is the healer type of agents, which would often exclude self observations and use only communication for their best rules. This is due to healers getting a bonus reward for healing a friendly unit, allowing rules that move near the intended actions of friendly units regardless of self hitpoints to quickly gain a higher weighting.

The combination of all of these issues creates agents that generally have very poor rules in RS_{int}. In all of these experiments the rules in RS were still built from the same set of observations of $Obs \backslash Obs_{int}$ and so one might expect that rules in RS should still be as effective as those of the agents without shout-ahead in Table 1, providing a lower bound on how poor agent performance can become with poor RS_{int} rules. This is not the case as both rule sets are used together in

Table 4
Performance of agents with friendly observations added to Obs_{coop}

Exp.	Best Gen.	Win/Loss/Tie
1	59	12/61/27
2	55	13/66/21
3	26	21/73/6
4	41	25/61/14
5	40	25/58/17
6	54	28/70/2
7	55	29/61/10
8	48	31/64/5
9	53	31/63/6
10	57	36/58/6
Av.	48.8	25.1/63.5/11.4

the game, and one may influence the reinforcement rewards of the other. For example an agent using a rule from RS may not perform as well due to other agents using poor rules from RS_{int} that same turn, resulting in potentially good rules in RS still being weighted low. In addition as explained in Section 3, when an agent is choosing between executing a rule from RS or a rule from RS_{int}, even if the rule from RS_{int} has a lower weight it may still be chosen. Thus the agents in this set of experiments will still regularly execute these poor rules, leading to the poor experimental results we found.

6.3.2. Evaluating the inclusion of Obs_{team}

In Table 4 we see the results of adding observations about friendly units to Obs_{coop}. Overall this change has led to the creation of agent teams that are only slightly worse on average than those of the experiments using shout-ahead in Table 1 (25 percent wins on average versus 30 percent).

Looking at the individual agents in each experiment shows that the inclusion of observations about friendly units has created a greater preference for movement over attacking, and in the lowest scoring experiments the highest weighted rules in RS_{int} include few or no attack rules. In addition these movement rules often require both a communicated intention of another agent to move nearby, and also for another friendly unit to already be present, as seen in the following rule which is the highest weighted rule in RS_{int} of the second melee agent of experiment 3 from Table 4.

IF $AdjacentFriendlyIsHurt(x) \wedge$
$CPMovingToAdjLoc(x) \wedge$
$CPAttackingAdjLoc(x) \wedge$
THEN $Move(x)$

This rule requires that there is already an injured friendly unit adjacent to the tile being considered for moving to, an agent has communicated an intention to move to a different adjacent tile, and finally an agent is attacking an enemy that is adjacent to the considered tile. When these conditions are met it is much more likely that there will be multiple friendly units adjacent to the tile at the end of the turn. This leads to a grouping behavior, where the agents move as a large group or several smaller groups rather than acting more individually. While this behaviour often leaves agents in a good defensive position, this was not enough to overcome the fact that enemy units were then the initiators of most battles, and were able to choose more advantageous terrain for themselves if it was available for them to attack from (the human created build-in AI is rather good at that).

Though this limits how effective the melee and ranged units can be, healer units benefited greatly from the inclusion of observations about friendly units. In our initial experiments, a healer agent would use communicated intentions to know where friendly units would move during the turn, and through that information position itself so that it would be besides a friendly unit at the end of the turn, restoring hitpoints at the beginning of the next turn. With the inclusion of observations about friendly units, healers were able to create rules to prefer healing some friendly agents over other ones. The best healers in these experiments had rules that allowed them to first favour injured leaders and units near the leader, decreasing the chance of the leader being defeated, as well as increasing the chance of healing multiple friendly units at the same time thanks to the grouping behaviour other agent types had.

As with the experiments using self observations, the agents in the experiments from Table 4 did create some rules that used no communicated intentions at all, but they did not suffer the same decrease in wins as all observations about friendly units may apply many times throughout an entire game unlike the self observation for full hitpoints. In addition, there are no observations that could be contradictory, and so these agents do not have the issue of non-functional rules that we described earlier. While the melee and ranged agents appear to perform worse due to attacking less, this is mostly balanced with the benefits of better healer units.

6.3.3. Evaluating inclusion of Obs_{enemy}

Table 5 shows the results of adding observations about enemy units to Obs_{coop}. This addition has al-

Table 5
Performance of agents with enemy observations added to Obs_{coop}

Exp.	Best Gen.	Win/Loss/Tie
1	59	24/52/24
2	37	25/67/8
3	45	29/65/6
4	27	33/61/6
5	60	35/51/14
6	58	36/64/0
7	38	49/47/4
8	38	49/42/9
9	47	53/46/1
10	60	60/38/2
Av.	46.9	39.3/53.3/7.4

lowed for agents to become much more effective at winning games, showing an average that is nearly 10% higher, as well as a best experiment with a 20% increase in wins and a team that is able to beat the default AI of Battle for Wesnoth much more often than loosing against it. Contrasting to the defensive behaviour created by combining communication with observations about friendly units, the inclusion of observations about enemies in this series of experiments has created teams of agents with strategies that are much more offensive, yet still often leader focused.

With observations about friendly units agents were able to tell which friendly unit was the leader, and that observation became a heavily used one in the best performing experiments of Table 4. Our agents that have observations about the enemy are instead able to know if an enemy threatens the leader, which is heavily used by the best teams in the experiments of Table 5 and helps the leader survive longer, this time by eliminating threats rather than blocking their access to the leader.

Favouring attacks on enemies that threaten the leader is not consistent among all unit types though, especially in the best scoring team of this experimental series. In experiment 10 from Table 5 melee agents have developed a strategy that focuses on pushing forward to the enemy leader, as seen in the following rule which is the highest weighted rule in RS_{int} of both top scoring strategies for melee agents.

IF $CPAttackingLocation(x) \wedge$
$ReachableLocIsClosestToEnLeader(x)$
THEN $Attack(x)$

This rule will have the agent attack a target enemy if another agent has communicated an intention to attack as well, and that enemy is the closest to the enemy leader out of those that could potentially be attacked. The rule also doubles as one that favours attacking the enemy leader over other enemies, as the leader is obviously the closest to itself. Ranged agents on the other hand favour attacking threats to the leader first, as seen in the top rule of the second best strategy for ranged agent.

IF $CPAttackingLocation(x) \wedge$
$EnThreatensLeader(x)$
THEN $Attack(x)$

The above rule will have the agent attack a target enemy if another agent has communicated the intention to attack, and that target enemy could possibly attack the friendly leader on its next turn. Attacking such an enemy reduces the number of attacks made on the friendly leader, allowing for it to survive longer. Often the same enemy meets the conditions of both the melee and ranged agents, resulting in a grouping behaviour similar to what was seen in the agents using observations about friendly units, but one which relied on attacking rules rather than movement ones.

It was initially expected that the healer units would suffer from the inclusion of enemy observations, as they lack the observations about friendly units that allowed them to better focus on healing damaged allies. Healers are also poor fighters that are generally on the losing end of a battle with melee and ranged units, yet they would have to attack more often due to the inclusion of these enemy observations. This appears to be what happened with the agents of experiment 1 in Table 5 where the problem of new observations overtaking communication is seen, again, and many of the healer's rules lack communicated intentions, resulting in healers that often attack enemies alone and die earlier in a game.

This did not happen with the healers in experiment 10 of Table 5, which use communicated intentions just as heavily as the other agents of that experiment. It is because of this communication that these healers do not die as quickly as those of experiment 1, even though their rules still use enemy observations. This improvement can in part be attributed to the order that agents take their actions in. First all units communicate their intended moves, then in order the leader, melee agents and ranged agents act, with healers acting last.

These particular agents have evolved so heavily towards cooperating on attacks that before a healer unit can attack, the target is often already dead due to multiple attacks from the melee and ranged units, and even the leader. This would then result in the healer's attack rule no longer being executed as the target no longer exists, and so the healers tend to use more movement rules. During a game, these healers will often move when possible to tiles that are adjacent to where enemies are, and in particular those with low health, are on disadvantaged terrain, or threaten the leader. Even though they themselves may not attack, their own communicated intention of the move increases the chance that a friendly unit will attack earlier in the turn. This also means the move will leave them besides that assisting unit ready to heal them on the next turn and restoring some of the hitpoints they lost from the attack.

These results further show that shout-ahead can make large improvements in creating agents that can compete well with AI systems that were created through other means. Through the lower performance seen with the inclusion of self observations, and the higher performance with the inclusion of observations about enemies we have also shown that while additional information may improve an agent using shout-ahead, that information must be chosen carefully. Not all information is beneficial, and some may severely disrupt the benefits that shout-ahead provides. On the other hand, if the information included is chosen carefully, there is the potential for large improvements. While the results using Obs_{self} were a surprise to us, the fact that using Obs_{enemy} improves the performance of agents is not so surprising: the communicated intentions naturally condense the views of the other friendly agents into the communicated action, but the enemies are not communicating, so that observations about them really add to the world view of an agent. Including other types of observations beyond communication also allows for the creation of teams with different play styles which players could then choose between separately from difficulty, analogous to the different AI strategies a player can choose from in many of the current generation of strategy games.

7. Related work

Naturally, there are a lot of papers about cooperation between agents and learning of cooperative behavior. We have cited a few of them in the introduction (i.e. [9,24]), but there are much more covering different agent architectures and many learning methods, including hybrid approaches. For example, [5] presents a hybrid method using reinforcement learning and a genetic algorithm to learn a real-time traffic control system. Even in the area of our application, computer games, there are several works dealing with learning of behavior and using hybrid techniques. But as stated in the introduction, there are very few works about learning of behavior of agents and communication between agents. And quite a number of the works that combine these topics are about how communication can improve the process of learning, so essentially about cooperative learning (see, for example, [2]).

The work nearest to the work presented in this paper is [1]. In this work, Aras et al. look at reinforcement learning for agents to win simple cooperative games (more precisely a modified battle of the sexes game with 3 actions per agent and not really an environment). The communication consists of simple messages that do not really contain more than just their existance and the sender. Obviously, our work tackles a much more complex application and uses messages with content. Also of a more theoretic nature and especially concerned with the cost of sending messages is [12]. This work focusses on the trade-off between cost of communication and the quality of the communicated information and, like other works of this group, is in the context of planning for a group of agents, which is not of real interest in our application (since due to the changing teams our agents have to be reactive and do not follow off-line learned plans). Consequently, the content of communication is about the state of the environment and not the intentions of other agents.

With regard to our application area, computer games, learning has become a rather hot topic over the last few years. But most of the known approaches learn on the player and not the unit level, which naturally means that communication is not an issue, but also means less possibility of reuse. For example, [18] uses an evolutionary algorithm for learning a single AI for the real-time strategy game Wargus. As another example, in [3] a central control for the game Civilization II is learned on-line by doing Monte-Carlo simulations of a limited length whenever a decision needs to be made. No real agent behaviors are learned, in fact there is no transitioning of what is learned to the next game (and naturally no cooperative strategies for units are learned). CARL (see [19]) combines reinforcement learning with case-based reasoning and a three-layered

agent architecture, where only the middle layer is performing learning, using reinforcement learning to determine what case to use to guide the decision making.

Examples for learning AIs that have to deal with only one unit are [11] and [4] which learned behaviors for the player in (Ms.) Pac-Man. [11] used evolutionary learning to learn a set of parameters representing probabilities for situation-action pairs. [4] compared an evolutionary algorithm with a reinforcement learning method, finding that the evolutionary method outperformed the reinforcement learning method. [6] evolved a neural network based controller for the ghost in Ms. Pac-Man. Evolving neural network based controllers for units in a game was already successfully proposed in [20] and improved in [13], although human help selecting the training scenarios was needed. Human guidance was also used in [22] to create human-like bot behavior for unreal tournament bots, in form of logs allowing to observe human game play. [17] combines evolutionary learning with reinforcement learning for a hand-to-hand combat game, but works on the Q-value matrices as representation of behavior.

8. Conclusion and future work

We presented an agent architecture (shout-ahead) and a hybrid learning method for it that allows agents to communicate their intentions to other agents. The architecture uses 2 sets of rules with weights that are learned via reinforcement learning and the weights are used to determine which rule set is responsible for the final decision an agent makes. Our evaluation of this idea in the context of learning cooperative behaviors for units in a turn based strategy game showed that the availability of the shout-ahead intentions improved the performance of the learned agents substantially and overall the learning of cooperative behaviors represents a way to create out of a single developer implemented AI strategy a wide variety of opponents of various quality. But our experiments also showed that the selection of what predicates should be used in addition to communicated intentions in the second rule set can have quite an effect on the performance of the learned behaviors.

There are several interesting directions for future work. Within Battle for Wesnoth, as already mentioned, there are several possibilities to improve the guidance for the learning by adding more knowledge to the rewards and the fitness of an agent. This would target the creation of very strong agents to be used in opponent teams for very strong and experienced human players. An evaluation of what different types of players think about the opponent teams created by learning is also an interesting piece of future research. Naturally, applying the shout-ahead approach to other kinds of applications for cooperative teams is also a direction we want to look into.

References

[1] R. Aras, A. Dutech and F. Charpillet, Cooperation through communication in decentralized Markov games, in: *Proc. AISTA'2004*, Luxembourg, 2004, http://hal.inria.fr/docs/00/03/60/77/PDF/aras_aista04.pdf.

[2] H. Berenji and D. Vengerov, Learning, cooperation, and coordination in multi-agent systems, Technical Report IIS-00-10, Intelligent Inference Systems Corp., 333 W. Maude Avennue, Suite 107, Sunnyvale, CA 94085-4367, 2000.

[3] S.R.K. Branavan, D. Silver and R. Barzilay, Learning to win by reading manuals in a Monte-Carlo framework, in: *Proc. 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, 2011, pp. 268–277.

[4] P. Burrow and S.M. Lucas, Evolution versus temporal difference learning for learning to play Ms. Pac-Man, in: *Proc. CIG-09*, Milan, 2009, pp. 53–60.

[5] M.C. Choy, D. Srinivasan and R.L. Cheu, Cooperative, hybrid agent architecture for real-time traffic signal control, *IEEE Trans. Sys. Man and Cybern. Part A: Systems and Humans* **33**(5) (2003), 597–607.

[6] J.-Y. Dai, Y. Li, J.-F. Chen and F. Zhang, Evolutionary neural network for ghost in Ms. Pac-Man. in: *Proc. ICMLC-2011*, Hong Kong, 2011, pp. 732–736.

[7] R. Davis and R.G. Smith, Negotiation as a metaphor for distributed problem solving, *Artificial Intelligence* **20** (1983), 63–109.

[8] J. Denzinger and S. Ennis, Improving evolutionary learning of cooperative behavior by including accountability of strategy components, in: *Proc. MATES 2003*, Erfurt, 2003, pp. 205–216.

[9] J. Denzinger and M. Fuchs, Experiments in learning prototypical situations for variants of the pursuit game, in: *Proc. ICMAS-96*, Kyoto, 1996, pp. 48–55.

[10] J. Denzinger and T. Offermann, On cooperation between evolutionary algorithms and other search paradigms, in: *Proc. CEC-99*, Washington, 1999, pp. 2317–2324.

[11] M. Gallagher and A. Ryan, Learning to play Pac-Man: An evolutionary, rule-based approach, in: *Proc. CEC'03*, 2003, Canberra, pp. 2462–2469.

[12] C.V. Goldman and S. Zilberstein, Optimizing information exchange in cooperative multi-agent systems, in: *Proc. AAMAS-03*, Melbourne, 2003, pp. 137–144.

[13] I. Karpov, V. Valsalam and R. Miikkulainen, Human-assisted neuroevolution through shaping, advice and examples, in: *Proc. GECCO 2011*, Dublin, 2011, pp. 371–378.

[14] V.R. Lesser and D.D. Corkill, Functionally accurate, cooperative distributed systems, *IEEE Transactions on Systems, Man and Cybernetics* **SMC-11** (1981), 81–96.
[15] L. Panait and S. Luke, Cooperative multi-agent learning: The state of the art, *JAAMAS* **11**(3) (2005), 387–434.
[16] S. Paskaradevan and J. Denzinger, A hybrid cooperative behavior learning method for a rule-based shout-ahead architecture, in: *Proc. IAT 2012*, Macao, 2012, pp. 266–273.
[17] L. Pena, S. Ossowski, J.M. Pena and S.M. Lucas, Learning and evolving combat game controllers, in: *Proc. CIG 2012*, Granada, 2012, pp. 195–202.
[18] M. Ponsen, Improving adaptive game AI with evolutionary learning, Master's thesis, Delft University of Technology, 2004.
[19] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell and A. Ram, Transfer learning in real-time strategy games using hybrid CBR/RL, in: *Proc. IJCAI-07*, Hyderabad, 2007, pp. 1041–1046.
[20] K. Stanley, B. Bryant and R. Miikkulainen, Evolving neural network agents in the NERO video game, in: *Proc. CIG-05*, Colchester, 2005, pp. 182–189.
[21] R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 1998.
[22] B. Tastan and G. Sukthankar, Learning policies for first person shooter games using inverse reinforcement learning, in: *Proc. AIIDE 2011*, Palo Alto, 2011, pp. 85–90.
[23] D. White, Battle for Wesnoth, http://www.wesnoth.org/ (as seen on 5.3.2012).
[24] C.H. Yong and R. Miikkulainen, Cooperative coevolution of multi-agent systems, Technical Report AI07-338, Department of Computer Sciences, The University of Texas at Austin, 2001.