Relational Macros for Transfer in Reinforcement Learning

Lisa Torrey[1], Jude Shavlik[1]
, Trevor Walker[1], and Richard Maclin[2]

[1] University of Wisconsin, Madison WI 53706, USA
[2] University of Minnesota, Duluth, MN 55812, USA

Abstract. We describe an application of inductive logic programming to transfer learning. Transfer learning is the use of knowledge learned in a source task to improve learning in a related target task. The tasks we work with are in reinforcement learning domains. Our approach transfers relational macros, which are finite-state machines in which the transition conditions and the node actions are represented by first-order logical clauses. We use inductive logic programming to learn a macro that characterizes successful behavior in the source task, and then use the macro for decision-making in the early learning stages of the target task. Using experiments in the RoboCup simulated soccer domain, we show that this transfer method provides a substantial head start in the target task.

1 Introduction

Knowledge transfer is an inherent aspect of human learning. When we learn to perform a task, we rarely start from scratch; we recall relevant knowledge from previous learning experiences and apply it. Transferring knowledge this way helps us to master new tasks more quickly.

Machine learning techniques are often designed to address isolated learning tasks. However, many machine learning domains contain multiple related tasks. *Transfer learning* approaches take advantage of these relationships, using knowledge learned in a *source task* to speed up learning in a related *target task*. Algorithms that allow successful transfer are steps towards making machine learning as adaptable as human learning.

One area in which transfer is often desirable is *reinforcement learning* (RL), since standard RL algorithms can require long training times. The RL domain that we use in this work is the simulated soccer project RoboCup [6]. In Section 2 we give an overview of RL and the RoboCup domain.

Several algorithms for transfer in domains like RoboCup have been proposed, some of which we discuss in Section 3. In our own recent work ([16]), we introduce an approach that transfers skills using inductive logic programming (ILP), where a *skill* is a type of action that the RL agent uses. In this paper, we extend that approach by transferring *strategies*, which are action plans that may require several skills. We continue to use ILP to learn strategies, and we represent them with a structure that we call a *relational macro*.

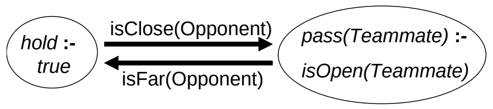

Fig. 1. A possible strategy for the RoboCup game KeepAway, in which the RL agent in possession of the soccer ball must execute a series of *hold* or *pass* actions to prevent its opponents from getting the ball. The rules inside nodes show how to choose actions. The labels on arrows show the conditions for taking transitions. Each node has an implied self-transition that applies by default if no exit transition does.

A relational macro is a finite-state machine (FSM) that uses first-order logic for decision-making. An FSM is a behavior model consisting of a set of nodes and transitions. To use a macro, an RL agent takes transitions to move between nodes that represent internal states and chooses actions to take in each node. Its choices are determined by first-order logical clauses. Figure 1 shows a simple example of a relational macro and Section 4 provides more details on how a macro is executed.

We use inductive logic programming (ILP) to learn macros because domains like RoboCup are inherently relational. To our knowledge, fully relational RL approaches have not yet been successfully applied in domains as complex as RoboCup. However, as we showed with skill transfer, relational information can be successfully transferred between RoboCup tasks. Therefore we continue to use ILP in this approach, describing source-task behavior in first-order logic.

Relational-macro transfer begins by examining existing source-task episodes and analyzing them to learn a successful strategy in the form of a macro. Section 5 describes our algorithm for this learning stage. There are several possible ways to use the macro to improve learning in the target task; we use it to demonstrate the successful strategy, as described in Section 6. After a short demonstration period that gives the target-task learner a head start, we continue learning the task with standard RL.

2 Reinforcement Learning in RoboCup

In reinforcement learning [13], an agent navigates through an environment trying to earn rewards or avoid penalties. The environment's state is described by a finite number of features, and the agent takes actions to cause the state to change. In one common form of RL called Q-learning [18], the agent learns a Q-function to estimate the value of taking an action from a state. An agent's *policy* is typically to take the action with the highest Q-value in the current state, except for occasional exploratory actions. After taking the action and receiving some reward, the agent updates its Q-value estimates for the current state.

Stone and Sutton [10] introduced RoboCup as an RL domain that is challenging because of its large, continuous state space and non-deterministic action effects. The RoboCup project [6] has the overall goal of producing robotic soccer teams that compete on the human level, but it also has a software simulator for research purposes. Since the full game of soccer is quite complex, researchers have

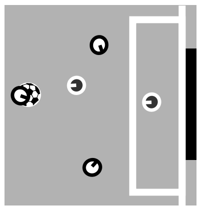

Fig. 2. Snapshot of a 3-on-2 BreakAway game. The attacking players have possession of the ball and are maneuvering against the defending team towards the goal.

developed several simpler games within the RoboCup simulator. See Figure 2 for a snapshot of one of these games.

In M-on-N BreakAway, the objective of the M reinforcement learners called *attackers* is to score a goal against $N - 1$ hand-coded *defenders* and a *goalie*. The game ends when they succeed, when an opponent takes the ball, when the ball goes out of bounds, or after a time limit of 10 seconds. The learners receive a $+1$ reward if they score a goal and 0 reward otherwise. The attacker who has the ball may choose to move (ahead, away, left, or right with respect to the goal), pass to a teammate, or shoot (at the left, right, or center part of the goal).

RoboCup tasks are inherently multi-agent games, but a standard simplification is to have only one learning agent. This agent controls the player currently in possession of the ball, switching its consciousness between players as the ball is passed. Players without the ball follow simple hand-coded policies that position them to receive passes.

Table 1 shows the state features for BreakAway, which mainly consist of distances and angles between players. They are represented in logical notation only for convenience later; our RL algorithm uses the grounded versions of these predicates in a fixed-length feature vector. Capitalized atoms indicate typed variables, while constants and predicates are uncapitalized. The attackers (labeled *a0*, *a1*, etc.) are ordered by their distance to the agent in possession of the ball (*a0*), as are the non-goalie defenders (*d0*, *d1*, etc.).

Our RL implementation uses a $SARSA(\lambda)$ variant of Q-learning [12] and employs a support vector machine for function approximation [5]. The exploration rate begins at 2.5% and decays exponentially over time. Stone and Sutton [10] found that discretizing the continuous features into boolean interval features called *tiles* is useful for learning in RoboCup; we follow this approach and create 32 tiles per feature.

Agents in the games of 2-on-1, 3-on-2, and 4-on-3 BreakAway take approximately 3000 training episodes to reach their maximum performance in our system. These three games are similar, but the changes in the numbers of attackers and defenders affect the optimal policy substantially. The most substantial difference is between 2-on-1 and the others, since there are no mobile defenders on the goalie's team in 2-on-1 BreakAway. However, the tasks do have the same objective and can be expected to require similar strategies, which makes relational macros an attractive technique for transferring between them.

Table 1. The features that describe a BreakAway state.

distBetween(a0, Player)	distBetween(Attacker, goalCenter)
distBetween(Attacker, ClosestDefender)	distBetween(a0, GoalPart)
angleDefinedBy(Attacker, a0, ClosestDefender)	timeLeft
angleDefinedBy(topRight, goalCenter, a0)	distBetween(Attacker, goalie)
angleDefinedBy(GoalPart, a0, goalie)	angleDefinedBy(Attacker, a0, goalie)

3 Related Work in Transfer Learning

The goal in transfer learning is to speed up learning in a target task by transferring knowledge from a related source task. One straightforward way to do this in reinforcement learning is to begin performing the target task using the learned source-task models. Taylor et al. [15] use this type of transfer method, which we refer to as *model reuse*.

Another approach that has been proposed is to follow source-task policies during the exploration steps of normal RL in the target task, instead of doing random exploration. This approach is referred to as *policy reuse* and is performed by Fernandez and Veloso [3].

Our previous work includes a method called *skill transfer* [16]. In skill transfer, we learn rules with ILP that indicate when the agent chooses to take single source-task actions. We call these concepts *skills*, and we use them as *advice* for the target task. The advice places soft constraints on the target-task solution, learned by a support vector machine, that can be followed or ignored according to how successful they are.

There are also approaches for transferring multi-step action sequences, performed by Perkins and Precup [7] and Soni and Singh [8]. Known as *options*, these sequences have their own internal Q-functions that are followed until they reach a termination state. The target-task learner treats options as alternative actions, which means option transfer is related to hierarchical RL [1].

One other related topic is relational reinforcement learning [14], in which state descriptions and learned models use first-order logic. We work with traditional domains that use fixed-length feature vectors, because most RL domains are still designed that way. However, relational RL clearly provides opportunities for transferring concepts in first-order logic. Driessens et al. [2] and Stracuzzi and Asgharbeygi [11] point out some of these opportunities.

We propose to perform transfer by learning relational macros and using them to demonstrate successful behavior in the target task. Our approach is related to several of the methods described above. It could be viewed as a type of model reuse that creates an abstract version of the source-task model instead of reusing it directly. Like skill transfer it uses ILP, but it involves multi-step strategies instead of single actions. It shares the idea of transferring sub-policies with option transfer, but an option traditionally represents a single policy while a macro contains a different one at each node.

4 Executing a Relational Macro

We have defined a relational macro as a finite-state machine [4]. An FSM models the behavior of a system in the form of a directed graph. The nodes of the graph represent states of the system, and in our case they represent internal states of the agent in which independent policies apply.

The policy of a node can be to take a single action, such as *move(ahead)* or *shoot(goalLeft)*, or to choose from a class of actions, such as *pass(Teammate)*. In the latter case a node has first-order logical clauses to decide which grounded action to choose. An FSM begins in a start node and has conditions for transitioning between nodes. In a relational macro, these conditions are also sets of first-order logical clauses.

We refer again to the example macro in Figure 1. When executing this macro, a KeepAway agent begins in the initial node on the left. The only action it can choose in this node is *hold*. It remains there, taking the default self-transition, until the condition *isClose(Opponent)* becomes true for some opponent player. Then it transitions to the second node, where it evaluates the *pass(Teammate)* rule to choose an action. If the rule is true for just one teammate player, it passes to that teammate; if several teammates qualify, it randomly chooses between them; if no teammate qualifies, it abandons the macro and reverts to using the Q-function to choose actions. The receiving teammate then becomes the learning agent, and it remains in the *pass* node if an opponent is close or transitions back to the *hold* node otherwise.

Figure 1 is a simplification in one respect: each transition and node in a macro has an entire set of rules, rather than just one rule. This allows us to represent disjunctive conditions. When more than one grounded action or transition is possible (i.e. multiple rules fire), the agent obeys the rule that has the highest score. The score of a rule is the estimated probability that following it will lead to a successful game. We estimate these probability scores from source-task data.

5 Learning a Relational Macro

We learn a macro by analyzing source-task data. We assume that this data is available because we have previously learned the source task and stored the games generated during the learning process. The method by which the source task was learned is not particularly important, since the data we use only consists of states, actions, and rewards. However, it is important that the data include source-task games from early in the learning curve as well as later, so that there are examples of both good and bad games. In our system we include all 3000 games from the source-task learning curve.

Given this data, we use inductive logic programming (ILP) to characterize successful behavior in the source task. Specifically, we use a locally modified version of Aleph [9]. The Aleph algorithm selects an example, builds the most specific clause that entails the example within the provided language restrictions, and searches for a more general clause that maximizes a provided scoring function.

Table 2. Our algorithm for learning a relational macro from a source task.

```
Phase 1: Structure learning
    Collect games from source task
    Let $Pos$ = high-reward games
    Let $Neg$ = low-reward games
    Learn a node sequence that distinguishes $Pos$ from $Neg$
Phase 2: Ruleset learning
    Collect games $G_{good}$ that contain the macro sequence and are high-reward
    Collect games $G_{bad}$ that are low-reward
    For each node N in the macro sequence
        For each action A represented by node N
            Let $Pos = G_{good}$ states from node N that took action A
            Let $Neg = G_{good}$ states from node N that took action B $\neq$ A
                    $\cup\ G_{bad}$ states that ended with action A
            Learn a ruleset that distinguishes $Pos$ from $Neg$
    For each transition T in the macro
        Let $Pos = G_{good}$ states that took transition T
        Let $Neg = G_{good}$ states that could have taken transition T and did not
                $\cup\ G_{bad}$ states that ended with transition T
        Learn a ruleset that distinguishes $Pos$ from $Neg$
```

The scoring function we use is

$$F(1) = \frac{2 * Precision * Recall}{Precision + Recall}$$

because we consider both precision and recall to be important. We use both the heuristic and randomized search algorithms provided by Aleph.

Recall that a macro consists of a set of nodes along with rulesets for transitions and action choices. The simplest algorithm for learning a macro might be to provide Aleph with language restrictions that allow it to learn both the structure and the rulesets simultaneously. However, this would be a very large search space. To make the search more feasible, we separate it into two phases: first we learn the structure, and then we learn each ruleset independently. Each phase therefore has its own language restrictions, which we detail in the following sections. The overall algorithm is summarized in Table 2.

Note that one final step might be necessary if the actions and features in the source and target tasks are not identically named: a *mapping* from source-task names to target-task names, as in Torrey et al. [16, 17]. The tasks do not even need to be completely isomorphic, because we can set the language restrictions so that only source-task elements that have corresponding target-task elements are included in the macro.

5.1 Structure Learning

The first phase in our algorithm for learning a macro is the structure-learning phase. The objective is to find a sequence of actions that distinguishes successful games from unsuccessful games. The sequence does not need to separate the games perfectly, and indeed we should not expect it to, because it does not yet have any conditions on states. The structure only needs to provide a good starting point for the second phase.

The language restrictions for Aleph in this phase are as follows. Let the predicate *actionTaken(G, S_1, A, P, S_2)* denote that action A with argument P was taken in game G at step S_1 and repeated until step S_2. Aleph must construct a clause *macroSequence(G)* with a body that contains a combination of these predicates. The first predicate may introduce two new variables, S_1 and S_2, but the rest must use an existing variable for $S1$ while introducing another new variable S_2. In this way Aleph finds a connected sequence of actions that translates directly to a linear node structure.

We provide Aleph with sets of positive and negative examples, where positives are games with high overall reward and negatives are those with low overall reward. For BreakAway, this is a straightforward separation of scoring and non-scoring games. For tasks with more continuous rewards, we could set thresholds on the overall reward acquired during a game or use upper and lower percentiles.

We store all the clauses that Aleph encounters during its search that separate the positive and negative examples with at least 50% accuracy. After the heuristic and randomized searches finish, we take the sequence with the highest F(1) score as the macro structure.

For instance, suppose that the scoring BreakAway games consistently look like these examples:

```
Game 1: move(ahead), pass(a1), shoot(goalRight)
Game 2: move(ahead), move(ahead), pass(a2), shoot(goalLeft)
```

Assuming that the non-scoring games have different patterns than the examples above do, Aleph might learn the following clause to characterize a scoring game:

```
macroSequence(Game) :-
    actionTaken(Game, StateA, move, ahead, StateB),
    actionTaken(Game, StateB, pass, _, StateC),
    actionTaken(Game, StateC, shoot, _, gameEnd).
```

The macro structure corresponding to this sequence is shown in Figure 3. The policy in the first node will be to take a single action, *move(ahead)*. In the second node the policy will be to consider multiple *pass* actions, and in the third node the policy will be to consider multiple *shoot* actions. The conditions for choosing an action, and for taking transitions between nodes, are learned in the next phase.

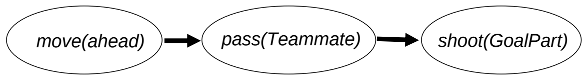

Fig. 3. The structure that corresponds to the example macro clause in Section 5.1.

5.2 Ruleset Learning

The second phase in our algorithm for learning a macro is the ruleset-learning phase. The objective is to describe when transitions and actions should be taken based on the RL state features. We learn a ruleset for each transition and each action independently, so that we perform several smaller, in-depth seaches rather than one large search.

The language restrictions for Aleph in this phase are as follows. There is one predicate for each state feature of the RL task (for BreakAway, these are in Table 1). To describe the conditions on state S under which a transition should be taken, Aleph must construct a clause *transition(S)* with a body that contains a combination of these predicates. To describe the conditions under which an action should be taken, the clause head depends on the action: for example, *pass(Teammate, S)* or *shoot(GoalPart, S)* or *move(Direction, S)*.

Because we learn the rulesets independently, the variables are local rather than global. Aleph may learn some rules in which the action argument is grounded, such as *pass(a1, S)*. In the case of the *move* action in BreakAway the action argument is always grounded, since the original state features do not include useful references to move directions. We could have defined additional predicates that did, but we chose to keep it simple and use only the original features. We do not learn rulesets for nodes in which the policy is to take a single grounded action, since there is no choice to make in these nodes.

We provide Aleph with sets of positive and negative examples, consisting of states in source-task games that were good and bad times to take the transition or action. Consider the macro structure in Figure 3; we will describe the action datasets for the *pass* node and the transition datasets for the transition from the *move* node to the *pass* node. Let G_{good} represent the set of high-reward source-task games that contain the macro sequence and let G_{bad} represent the set of low-reward source-task games.

In the action datasets for the *pass* node, the positive examples are states in G_{good} games that fall into that node. The negative examples are states in G_{bad} games in which the last step of the unsuccessful game was a *pass* action. Figure 4 illustrates some hypothetical examples.

In the transition datasets for the transition from the *move* node to the *pass* node, the positive examples are states in G_{good} games that fall into the *pass* node and for which the previous state fell into the *move* node. A negative example is a state in a G_{good} game that does not fall into the *pass* node even though the previous state fell into the *move* node. Other negative examples are states in G_{bad} games in which the last step of the unsuccessful game was a transition from the *move* node to the *pass* node. Figure 5 illustrates some hypothetical examples.

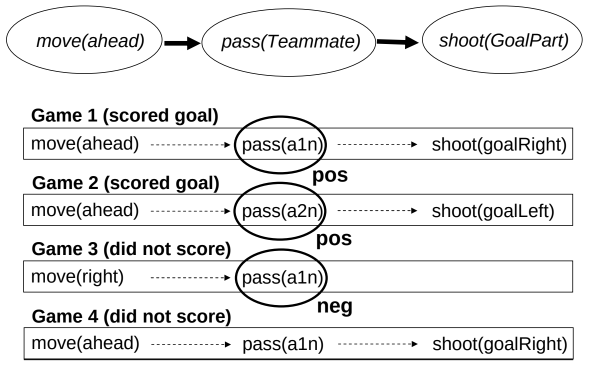

Fig. 4. Some training examples for *pass(Teammate)* rules in the second node of the pictured macro. The pass states in games 1 and 2 are positive examples. The pass state in game 3 is a negative example. The pass state in game 4 is not a good example because a later action may have been more responsible for the bad outcome.

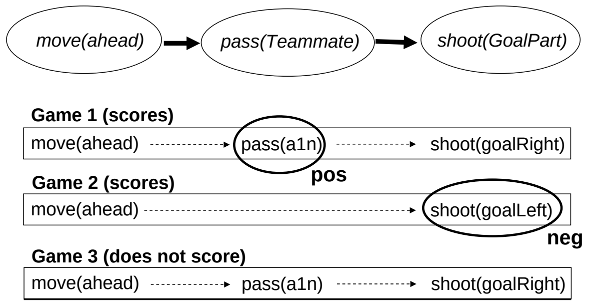

Fig. 5. Some training examples for the transition from *move* to *pass* in the pictured macro. The pass state in game 1 is a positive example. The shoot state in game 2 is a negative example. No state in game 3 makes a good example because a later step may have been more responsible for the bad outcome.

As in the first phase, we store all the clauses that Aleph encounters during the search that classify the training data with at least 50% accuracy. However, instead of selecting a single best clause as we did in the previous phase, we collect from these a ruleset for each transition and each action. We wish to have one strategy (one node structure), but there may be multiple reasons for making internal choices.

Our procedure for selecting which clauses are included in a ruleset is summarized in Table 3. We sort the rules by decreasing precision and walk through the list, greedily adding rules to the final ruleset if they increase the set's recall and do not decrease its F(1) score.

We assign each rule a score that may be used to decide which rule to obey if multiple rules fire while executing the macro. The score is an estimate of the probability that following the rule will lead to a successful game. We calculate

Table 3. Our procedure for selecting the final ruleset for one transition or action. Rules are added to the final set if they cover previously uncovered positive examples and do not decrease the overall score. The scoring function is the F(β) measure.

```
Let $R$ = all rules encountered with $> 50\%$ accuracy
$S = R$ sorted by decreasing precision on the training set
$T = \emptyset$
For each rule $r \in S$
    $U = T \cup \{r\}$
    If recall($U$) $>$ recall($T$) and score($U$) $\geq$ score($T$)
    Then $T = U$
Return $FinalRuleset = T$
```

this estimate by collecting source-task games that follow the rule and finding the fraction of these that end up successful.

After this phase the macro is complete. In the next section we describe how we use a macro to improve learning in the target task.

6 Transferring a Relational Macro

A relational macro describes a strategy that was successful in the source task. There are several ways we could use this information to improve learning in a related target task. One possibility is to use *advice*, as we did in skill transfer, to put soft constraints on the Q-learner that influence its solution. The benefit of this approach is its robustness to error: if the source-task knowledge is less appropriate to the target task than the user expected, the target-task agent can learn to disregard the soft constraints.

On the other hand, the advice-taking approach is conservative and somewhat slow to reach its full effect when the source-task knowledge is highly appropriate to the target task. Since we have a full strategy here rather than isolated skills, we might achieve good target-task performance more immediately by *demonstrating* the strategy in the target task and using it as a starting point for learning. This is a more aggressive approach, carrying more risk for negative transfer if the source and target tasks are not similar enough. Still, if the user believes that the tasks are similar enough, the potential benefits could outweigh that risk.

There are intermediate approaches that carry more moderate benefits and risks, such as using the macro as an option. However, in the interests of providing a contrasting method to skill transfer, we present the more aggressive demonstration method here.

We begin by performing a set of episodes in the target task in which we execute the macro strategy. In this demonstration period, we are generating examples of Q-values: each time the macro chooses an action because a high-scoring rule fired, we use the rule score as an estimate of the Q-value of the action. We also have Q-value estimates for other actions that were not chosen but for which rules fired. Finally, we can infer that actions for which no rules fired had very low Q-values, and we estimate them as zero. For each step of the demonstration we therefore have a Q-value estimate for each action, and using support vector regression we use these to learn an initial Q-function for the target task.

The demonstration period lasts for 100 games in our system. After it ends, we continue learning the target task with standard RL. This generates new Q-value examples in the standard way, and we combine these with the old macro-generated examples as we continue updating the Q-function. As the new examples accumulate, we gradually drop the old examples by randomly removing them at approximately the same rate that new ones are being added.

Since standard RL has to act mostly randomly in the early steps of a task, a good macro strategy can provide a large immediate advantage. The performance level of the demonstrated strategy is unlikely to be as high as the target-task agent can achieve with further training, unless the tasks are similar enough to make transfer a trivial problem, but the hope is that the learner can smoothly improve its performance from the level of the demonstration up to its asymptote. If there is limited time and the target task cannot be trained to its asymptote, then the immediate advantage that macros can provide may be even more valuable in comparison to methods like skill transfer.

7 Experimental Results

We present results from transfer experiments in the RoboCup domain. To test our approach, we learn a macro from data acquired while training 2-on-1 BreakAway and transfer it to both 3-on-2 and 4-on-3 BreakAway. We learn the source task with standard RL for 3000 games, and then we train the target tasks for 3000 games to show both the initial advantage of the macros and the behavior as training continues.

Figures 6 and 6 show our results in 3-on-2 and 4-on-3 BreakAway. We compare our approach against Q-learning as well as two related transfer methods: model reuse [15] and skill transfer [16]. Each curve in the figure is an average of 25 runs and has points smoothed over 500 games to control for the high variance in the RoboCup domain. For the transfer algorithms, there are five target-task runs generated from each of five source-task runs, to allow for variance in both stages of learning.

The macros our algorithm learned from the five source runs all had similar structures. The most common version is shown in Figure 8. In one of the runs the initial *pass* node was not included, and the ordering of *shoot(goalRight)* and *shoot(goalLeft)* varied. The presence of two *shoot* nodes may seem counterintuitive, but it appears that the RL agent uses the first shot as a feint to lure the goalie in one direction, counting on an automated teammate to intercept the shot before it reaches the goal. When it does, the learning agent switches to the teammate in possession of the ball and performs the second shot, which is actu-

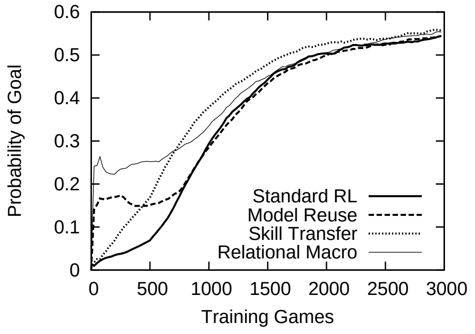

Fig. 6. Probability of scoring a goal in 3-on-2 BreakAway, with Q-learning and with three transfer approaches that use 2-on-1 BreakAway as the source task.

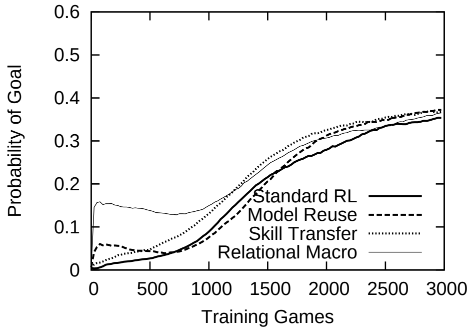

Fig. 7. Probability of scoring a goal in 4-on-3 BreakAway, with Q-learning and with three transfer approaches that use 2-on-1 BreakAway as the source task.

ally intended to score. This tendency of RL agents to use actions in unintended ways is an indication of the difficulties that can arise when learning relational concepts from RL data.

Agents in 2-on-1 BreakAway reach a performance asymptote during their 3000 games in which they score in approximately 70% of the episodes. The macros that we learned from the 2-on-1 source runs, when executed in 2-on-1 BreakAway, score in approximately 50% of the episodes. The macros therefore capture the majority of the successful behavior of the source task, though they do not describe it completely.

All of the transfer algorithms speed up learning in comparison to Q-learning, but the benefits they provide are different. Model reuse and relational macros both provide an advantage in the early performance of the target-task learner. The macro approach produces a larger advantage in these scenarios than model reuse does, and it scales better as the distance between the source and target grows. Skill transfer provides no initial benefit, but then develops a steady advantage over Q-learning. During the middle section of the learning curve it performs slightly better than the macro approach before they all converge at the asymptote.

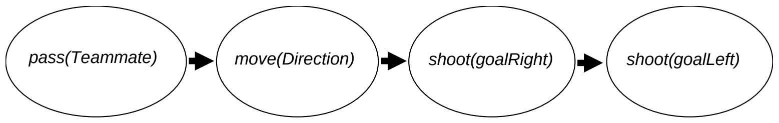

Fig. 8. One of the five macro structures learned from 2-on-1 BreakAway runs. There are between 10 and 20 rules associated with each transition and action, so those are not shown.

8 Conclusions and Future Work

Knowledge transfer in reinforcement learning is an interesting and challenging problem, and inductive logic programming is a powerful tool to apply to it. The use of ILP allows us to transfer the kind of information that humans might transfer: strategies with decisions in first-order logic. We describe an approach for transferring relational macros from a source task that gives the target-task learner a significant head start.

In our experiments, our approach speeds up learning in comparison to standard RL. It also performs consistently better than one related method, model reuse [15]. Compared to another related method, skill transfer [16], it produces significantly higher initial performance but slower learning later on. This suggests an intuitive tradeoff in transfer learning: approaches that allow fewer mistakes early in the learning process will have higher early performance, but may have slower learning rates.

In future work, we plan to look at alternative ways to apply relational macros in the target task. An ideal approach would keep most of the initial benefits of the aggressive demonstration method but alleviate some of the risks. This might make macro transfer applicable to source and target tasks that have less similar strategies. As it is, our approach should only be chosen if the user is confident that a source-task strategy is a reasonable approximation of a good target-task strategy.

Another interesting extension of this work would be to apply it in a fully relational RL approach. Our algorithm could be performed the same way, simply substituting a fully relational learner for propositional Q-learning, but this might also provide opportunities to extend our approach. In this context, our work could be viewed as a step toward extending relational reinforcement learning to challenging domains like RoboCup.

9 Acknowledgements

This research is supported by DARPA grant HR0011-04-1-0007 and by DARPA IPTO under contract FA8650-06-C-7606.

References

1. T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.
2. K. Driessens, J. Ramon, and T. Croonenborghs. Transfer learning for reinforcement learning through goal and policy parametrization. In *ICML Workshop on Structural Knowledge Transfer*, 2006.
3. F. Fernandez and M. Veloso. Policy reuse for transfer learning across tasks with different state and action spaces. In *ICML Workshop on Structural Knowledge Transfer*, 2006.
4. A. Gill. *Introduction to the Theory of Finite-state Machines.* McGraw-Hill, 1962.
5. R. Maclin, J. Shavlik, L. Torrey, and T. Walker. Knowledge-based support vector regression for reinforcement learning. In *IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005.
6. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.
7. T. Perkins and D. Precup. Using options for knowledge transfer in reinforcement learning. Technical Report UM-CS-1999-034, 1999.
8. V. Soni and S. Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *AAAI*, 2006.
9. A. Srinivasan. *The Aleph Manual*, 2001.
10. P. Stone and R. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *ICML*, 2001.
11. D. Stracuzzi and N. Asgharbeygi. Transfer of knowledge structures with relational temporal difference learning. In *ICML Workshop on Structural Knowledge Transfer*, 2006.
12. R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning 3*, pages 9–44, 1988.
13. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.
14. P. Tadepalli, R. Givan, and K. Driessens. Relational reinforcement learning: An overview. In *ICML Workshop on Relational Reinforcement Learning*, 2004.
15. M. Taylor, P. Stone, and Y. Liu. Value functions for RL-based behavior transfer: A comparative study. In *AAAI*, 2005.
16. L. Torrey, J. Shavlik, T. Walker, and R. Maclin. Relational skill transfer via advice taking. In *ECML*, 2006.
17. L. Torrey, T. Walker, J. Shavlik, and R. Maclin. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *ECML*, 2005.
18. C. Watkins. Learning from delayed rewards. Technical Report PhD Thesis, University of Cambridge, Psychology Dept., 1989.