Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

An Object-Oriented Approach to Reinforcement Learning in an Action Game

Shiwali Mohan and **John E. Laird**
Computer Science and Engineering
University of Michigan, Ann Arbor, MI 48105
{shiwali, laird}@umich.edu

Abstract

In this work, we look at the challenge of learning in an action game, Infinite Mario. Learning to play an action game can be divided into two distinct but related problems, *learning an object-related behavior* and *selecting a primitive action*. We propose a framework that allows for the use of reinforcement learning for both of these problems. We present promising results in some instances of the game and identify some problems that might affect learning.

Introduction

Traditionally, artificial intelligence (AI) has been used to control non-player characters to make games more interesting to a human player. This is a challenging problem in its own right. However, we hold a different view towards AI in games. We want to place an artificially intelligent agent in a game instead of a human player, and investigate what knowledge and learning capabilities are required of an agent to play it effectively. In this work, we have concentrated on a variant of a popular action game Super Mario brothers, called Infinite Mario.

Action games have been popular among gamers since the earliest video games were developed. The action genre emphasizes acquiring certain physical reactive skills and behaviors to progress in the game. Sometimes tactical and exploration challenges are incorporated, but most of the games are designed around high reaction speed and physical dexterity. Often, strategic planning is hard because of the time pressure to complete the game, motivating simple heuristic strategies. Action games have been popular with AI research community and have been studied in detail using rule based systems (Laird and van Lent, 2001) and static scripting (Spronck et al., 2006).

The contribution of this work is two-fold. First, through this work we characterize the learning problem in the action game, Infinite Mario, as a combination of *learning object-oriented behaviors* and *action selection*. Secondly, we propose a reinforcement learning framework that gives promising results on a subset of game instances.

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Related Work

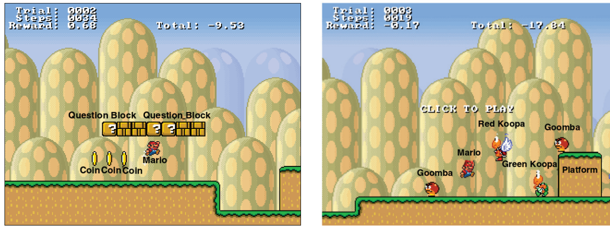
RL has been widely explored for its applicability in RTS games domain. Driessens (2001) combined RL with regression algorithms to generalize in the policy space. He evaluated his relational RL approach in two RTS games and RPGs (role playing games). Marthi et al. (2005) applied hierarchical RL (HRL) to scale RL to a complex environment. They learned navigational policies for agents in a limited real-time strategy computer game domain. Their action space consisted of high-level pre-programmed behaviors called partial programs, with a number of choice points that were learned using Q-learning. Ponsen, Spronck, and Tuyls (2006) employed a deictic state representation in HRL that reduces the state space complexity as compared to a propositional representation and allows the adaptive agent to learn a generalized policy. Their architecture is capable of transferring knowledge to unseen task instances and significantly outperforms flat learning.

Our work builds on the concepts introduced by the work done with RL in RTS games. Learning in strategy games concentrates on developing an optimal strategy to win the game through a search in the strategy space. These games differ from the action game genre, where at any time, quick decisions are to be made about competing goals and, in general, the emphasis is on finishing the game as fast as possible. Challenges presented by action games are considerably different from games that have been previously explored and we will discuss some of them in the following sections.

In the domain of action games, Diuk, Cohen, and Littman (2008) introduce Object-Oriented Markov Decision Processes, a representation that is based on objects in the environment and their relationships with each other. They conjecture that such representations are a natural way of describing many real-life domains and show that such a description of the world enables the agent to learn efficient models of the action game, Pitfall. Wintermute (2010) has explored agents that learn to play various Atari action games using predictive features generated by spatial imagery models of physical processes and reports improvements over RL agents using typical algorithms and state features. Our model free framework is considerably different, and requires little to no domain knowledge.

Infinite Mario has recently become a popular domain for AI research with the advent of the Mario AI competition.

Many agents have been written that are efficient at playing the game. Most of them are based on A* search for path finding and use detailed models of game physics and minimal learning. In contrast, we are looking at learning agents that start with little or no background knowledge about the physics or the terrain of the game.



(a) Level 0, Seed 121 (b) Level 1, Seed 121

Figure 1: Screenshots from Infinite Mario

Preliminaries

RL is a computational approach to automating goal-directed learning and decision making. It encompasses a broad range of methods for determining optimal ways for behaving in complex, uncertain and stochastic environments. Most current RL research is based on the theoretical framework of Markov Decision Processes (MDPs) (Sutton and Barto, 1998).

Markov Decision Processes

An MDP is defined by its state and action sets and by the dynamics of the environment. At each time step t, the agent observes the state of its environment, s_t, contained in a finite set S, and decides on an action, a_t, from a finite action set A. A time step later, the agent receives reward r_{t+1} and the environment transitions to the next state, s_{t+1}. The environment is assumed to have the *Markov Property*: its state and reward at $t+1$ depend only on the state and action at t and not on any past values of states and actions. Given any state and action, s and a, at time t the *transition* probability to possible next state, s', at time $t+1$ is:

$$P_{ss'}{}^{a} = Pr\{s_{t+1}|s_t = s, a_t = a\}$$

Similarly, given any state and action pair, s and a, at time t along with the next state, s', at time $t+1$ the expected value of the reward is given by:

$$R_{ss'}{}^{a} = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$$

The RL problem is finding a way of acting in the environment such that the reward accumulated is maximized. The way of behaving, or the *policy*, is defined as a probability distribution for selecting actions in each state: $\pi : S \times A \rightarrow [0, 1]$. The goal of the agent is to find a policy that maximizes the total reward received over time. For any policy π and any state $s\epsilon S$, the value of taking action a in state s under policy π, denoted $Q^\pi(s, a)$, is the expected discounted future reward starting in s and taking a, and following π from then on:

$$Q^\pi(s, a) = E_\pi\{r_{t+1} + \gamma r_{t+2} + ...|s_t = s, a_t = a\}$$

The *optimal* action-value function is:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

In an MDP, there exists a unique optimal value function, $Q^*(s, a)$, and at least one optimal policy, π^*, corresponding to this value function:

$$\pi^*(s, a) > 0 \iff a\epsilon \arg\max_a Q^*(s, a)$$

Many popular RL algorithms aim to compute Q^* (and thus implicitly π^*) based on an agent's experience of the environment. One of the most widely-used algorithm, SARSA, was introduced by Rummery and Niranjan (1994). It is based on the following update equation:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma Q(s', a')]$$

where $0 \leq \alpha \leq 1$ is a learning rate parameter and $0 \leq \gamma < 1$ is the discount factor for future rewards. SARSA (s, a, r, s', a') is an on-policy learning algorithm, it estimates the value of the same policy that it is using for control. All the agents we describe use SARSA for policy updates.

Semi-Markov Decision Processes

Semi-Markov decision processes (SMDPs) serve as the theoretical basis for many hierarchical RL approaches developed during the last decade to solve complex real world problems. In these hierarchical approaches, temporally extended and abstract actions need to be modeled. SMDPs may be considered a generalization of MDPs. Barto and Mahadevan (2003) define an SMDP as a continuous-time decision problem that is treated as a discrete-time system, where the system makes discrete jumps from one time at which it has to make a decision to the next.

Formally, an SMDP is a tuple $< S, B, T, R >$, where S is a set of states, and B is a set of temporally abstract actions. Given a state s and an abstract behavior b that is k time steps long, the *transition* probability to the next state, s' at $t + k$ is given by:

$$P^b_{ss'_k} = Pr(s_{t+k}|s_t = s, b_t = b)$$

$$R(r|s, b) = Pr(r_t = r|s_t = s, b_t = b)$$

P and R both obey Markov Property.

A policy is a mapping $\pi : S \rightarrow B$ from states to behaviors. Executing a behavior results in a sequence of primitive actions being performed. The value of the behavior is equal to the value of that sequence. Thus, if behavior b_t is initiated in state s_t and terminates sometime later in state s_{t+k} then the SMDP reward value r is equal to the accumulation of the one-step rewards received while executing b_t:

$$r = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{k-1} r_{t+k-1}$$

which gives identical state-value function as in MDPs. Since the value measure V^π for a behavior based policy π is identical to the value measure V^π for a primitive policy we know that π^* yields the optimal primitive policy over the limited set of policies that a hierarchy allows.

Problem Characterization

Infinite Mario, a variant of Nintendo's Super Mario Brothers, is an RL domain developed for the 2009 RL Competition. It is a side-scrolling game with destructible *blocks*, *enemies*, *fireballs*, *coins*, *chasms*, and *platforms* (Figure 1a and Figure 1b). It requires the player to move right towards the *finish line*, earning points and *powers* along the way by collecting *coins*, *mushrooms*, *fire flowers*, and killing *enemies* like the *Goomba*, *Koopa*, etc. The game is partially observable and the player can never perceive the complete game state. Like other action games, it requires the player to learn favorable interactions (*behaviors*) with the different elements (*objects*) of the game. For example, a seasoned player would be an expert at killing the *enemies* like *Goomba* by *stomping* on them, and would know that acquiring a *Mushroom* is beneficial to the game play. On reaching the *finish line* in Super Mario Brothers, the player progresses to the next level. However, in Infinite Mario, the game restarts at the same level on reaching the finish line.

Game State, Action and Score

Infinite Mario has been implemented on RL-Glue developed by Tanner and White (2009); a standard interface that allows connecting RL agents, environments, and experiment programs together.

- *State Observations*: The state observations from the framework have two components: the visual scene, which includes the stationary objects (*coins*, *blocks* etc), and the moving objects (*Mario*, *Goomba* etc). The visual scene is divided into tiles of equal size but different types, and is provided to the agent as an array. The scene changes as *Mario* progresses in the game.
- *Actions*: The primitive actions available to the agent are typical of a Nintendo controller. The agent can choose to move right or left or can choose to stay still. It can jump while moving or standing. It can move at two different speeds.
- *Score*: For every successful interaction with *coin*, *block*, *enemies* etc. the player is scored positively. The player is scored negatively on *dying* before reaching the *finish line* and gets a high positive score on reaching the *finish line*. The scores are directly translated to rewards for an RL agent. A small negative reward is given for every action taken in the game to encourage the agent to finish the game sooner. Note that the environment provides a composite reward for the *complete state* in the environment.
- *Levels*: The domain is capable of generating several instances of the game with great variability in difficulty, i.e. as the agent moves to a higher level, making good decisions becomes harder. This can be due to close interaction with many objects at the same time or having more constraints on how the agent can move (see Figure 1b). The lowest level, which is the easiest, is level 0.
- *Memory Agent*: The software includes a memory agent that we used for evaluation as a baseline. The agent learns through memorization in that it stores the sequence of actions (decided using some heuristics) it takes as it moves through the episode. This leads to good performance in a game instance because all the transitions are deterministic and the game always starts in the exactly the same state. However, the total reward earned by the agent is limited by its negligible exploration of the environment.

The Action Selection Problem

At each time step, the player faces an action selection problem - given the current state of the game, what action or sequence of actions would lead to a favorable outcome. RL techniques lead to goal oriented behaviors as they seek to maximize the reward signal generated by the environment. This reward is often related to goals. Past work on learning in complex environments, such as games, involves dividing a large problem into a series of smaller problems. To achieve the overall goal, these subproblems have to be solved. Solutions to these problems can be learned hierarchically, as shown in Dietterich (2000) and in Ponsen, Spronck, and Tuyls (2006) for strategy games.

For action games, overall goals are harder to define. A goal in an action game like Infinite Mario could be to reach the *finish line*. However, while achieving that goal, the agent has to target several smaller, local goals such as *killing the enemy*, *collecting coins*, *jumping over the pits*, *jumping on to the platforms* etc. These goals do not arise as subproblems of solving the complete game nor do they aid in reaching the *finish line*; they are independent of the overall goal of the game but have to be accomplished for a good score (and to refrain from *dying*). A successful play would involve being an expert at achieving all these goals while moving towards the finish line. The overall goal of *finishing* the game cannot be hierarchically divided into a static sequence of subgoals to be achieved. Often these goals suggest conflicting actions in the environment and their occurrence is independent of each other. We look to RL techniques to provide us with evaluations of actions suggested by competing goals, which are then used to select an appropriate action using some simple heuristics.

Properties of the Domain

The domain is highly complex in a variety of ways:

- *Continuity*: The domain is continuous, and the positions of objects on the visual scene are real-valued. A true, continuous state representation would lead to an infinitely large state space. A simple solution to this problem is to discretize the visual scene, by grouping spatially close states into a single state. However, this is to be approached with caution; learning the value function will be impossible if the discretization is not fine enough and functionally distinct states are merged together. On the other hand, an extremel y fine discretization will lead to an enormous state space.
- *High Dimensionality*: At any time step, the visual scene is occupied by a large number of objects and corresponding goals. The action selection problem given these multiple goals is combinatorial in the number of objects and is largely intractable. In a prior study on the game, John and Vera (1992) observed that a human expert concentrated

on interacting with objects that were very close to Mario. The spatial distance between the objects and *Mario* was indicative of which objects were important to interact with first, greatly constraining the action selection problem.

- *Dynamic*: There is a high degree of relative motion between objects in the game and although the domain is not real-time (the environment waits for an action from the agent before stepping), the state might change independently of the actions of the agent.
- *Determinism*: The environment is deterministic, however the state abstractions introduced to make learning tractable might introduce non determinism of a varying degree, leading to unpredictable outcomes and often, a failure to learn.
- *Partial Observability*: At any given time, the agent can perceive only a part of the complete state of the game. There is little reason to believe that parts of the state that are not visible to the agent might play an important part in action selection in Infinite Mario. However, as above, various abstractions applied to make learning in the game tractable can lead to partial observability.

Object-Oriented Design and Empirical Results

We are interested in an *object-oriented* view of the environment, where the agent learns *object-oriented behaviors* to achieve *object-related goals*. For example, an agent learns a behavior *tackle-enemy* associated with an object belonging to a class *enemy*, with the goal of *killing* the *enemy* by stomping on it. This approach is motivated in part, by a GOMS (Goals, Operators, Methods and Selection rules) analysis of Nintendo's Super Mario Brothers conducted in 1992 by John and Vera (1992). They demonstrated that the behavior of a Soar agent that used simple, hand coded heuristics formulated by taking knowledge explicit in the instruction booklet and by reasoning about the task, was predictive of a human expert playing the game. They also reported that the expert's behavior was highly object-oriented. The analysis was carried out at two levels: the *functional-level-operators* that constitute the *behaviors* associated with objects and the *keystroke-level-operators* or the primitive actions. Our implementation closely follows theirs, however we show that these *behaviors* can be learned by the agent by acting in the envirovment.

An *object oriented*, learning agent has two distinct but inter-dependent problems:

Learning a behavior

Through RL, the agent learns a set of primitive actions that define a *behavior* for every class of objects. These behaviors should form an ideal interaction with the object, culminating in achieving the goal associated with it and consequently earning a positive reward. For example, the behavior *grab-coin*, instantiated for a *coin* <*c*> ends a positive reward and absence of <*c*> in future observations. The positive reward received while executing *grab coin* guides learning the correct sequence of primitive actions.

The state representation we use is similar to those of Relational MDPs (Guestrin et al., 2003) with some differences. Similar to RMDPs, we define a set of classes $C = \{C_1, ..., C_c\}$. Each class includes a set of attributes $Attribute(C) = \{C.a1, ..., C.aa\}$, and each attribute has a domain $Dom(C.a)$. The environment consists of a set of objects $O = \{o_1, ..., o_o\}$, where each object is an instance of one class: $o \epsilon C_i$. The state of an object, $o.state$, is a value assignment to all its attributes. The formalism by Guestrin et al. (2003) describes the complete state of the underlying MDP as the union of the states of all its objects. We, on the other hand are interested in learning independent, object-related MDPs rather than learning the composite MDP. The composite MDP is combinatorial in the number of objects, hence learning it is intractable in Infinite Mario.

While learning an object-related MDP, we assume that other objects present in the observations do not affect the reward function (*isolated interaction* assumption). This is an over-general assumption and is met only in specific cases. This also implies that what we are trying to learn is essentially a partially observable MDP. Since the problem is framed as a POMDP, convergence of the value function is not guaranteed. However, in lower difficulty levels the assumption of isolated interaction holds and the agent is able to learn good policies.

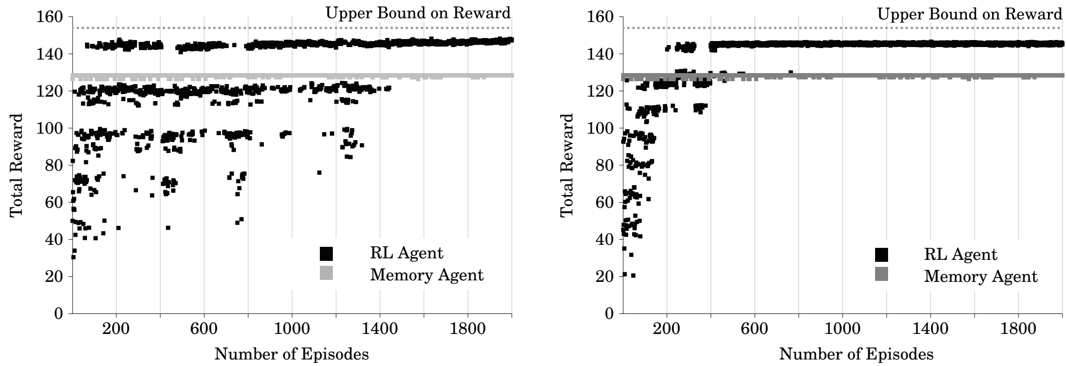
The action space is composed of the primitive actions available and the transitions are dependent on the dynamics of the game.

Selecting a primitive action

The selection problem arises when there are multiple, visible objects. Learned *behavior* for each of those objects may suggest conflicting primitive actions. There are multiple strategies to select an action. One of the strategies is to select an object first, and then execute the *behavior* corresponding to the object. The knowledge for object selection can be hand-programmed or learned using RL.

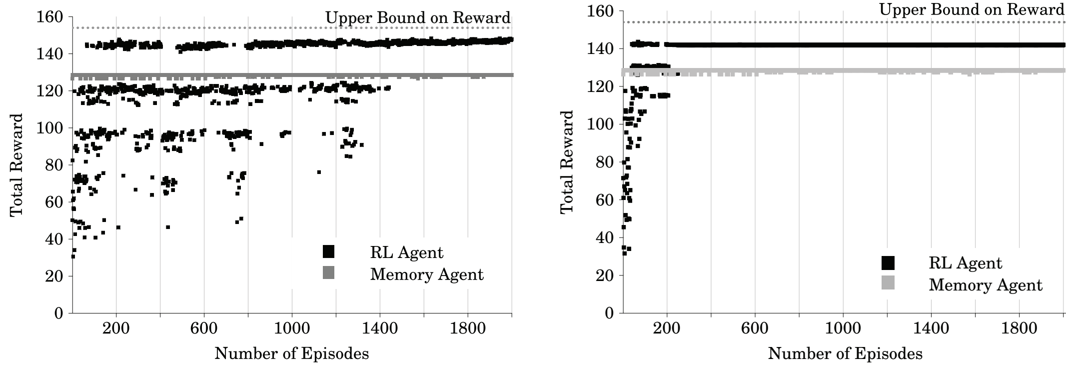
To learn object selection, we look at prior work done in hierarchical RL (Dietterich, 2000) which relies on the SMDP framework. Playing the game can be formalized as an SMDP, where object related *behaviors* are temporally extended actions and the agent learns the value of choosing a particular *behavior* over others, given a set of visible objects spatially close to *Mario*. For the state representation, we define a relation *is-close(x,Mario)* which is *true* if an object is spatially close to *Mario*. The 'closeness' relationship can be independently defined for each class of objects. The set of available behaviors form the temporally extended action space and the transitions are based on the presence or absence of objects. The task of playing the game can be formulated as an SMDP based on present objects, and a value function is learned that gives a sense of the 'goodness' of a behavior in a state.

These two problems affect each other. The 'availability' of a particular *behavior* depends on the presence of the corresponding object. The agents selects a *behavior* based on its past experience with similar states. Once a *behavior* is selected, the agents executes the *behavior* while learning more about the successful interaction with the correspond-



(a) Using hand coded object selection strategy, environmental rewards

(b) Using learned object selection, environmental rewards



(c) Using background spatial knowledge

(d) Using learned object selection, goal related rewards

Figure 2: Learning performance on Level 0, Game Seed 121 of Infinite Mario. The data is averaged over 10 runs of 2000 episodes each. The black data points indicate the average reward earned by our RL agents in a particular episode. The light grey data points indicate the average reward earned by the memory agent. The upper bound in the particular instance of the game is indicated by the dotted line.

ing object and the utility of selecting the *behavior* in the current state. This new knowledge, in turn, affects the selection when a similar state is encountered next.

The results are presented in Figure 2. All results have been averaged over 10 trials of 2000 episode each. The q-values are updated using SARSA with $\alpha = 0.3$. The action selection policy used is epsilon-greedy with $\epsilon = 0.1$ and an epsilon-decay-rate of $\gamma = 0.99$. Since the domain to too complex to be studied mathematically, the optimal policy is hard to analyze. To get an idea of how good our agents are performing, we estimated the upper bound on the reward that can be achieved in the level by counting the reward given for every object in the level. The optimal policy will earn strictly less than the upper bound.

Figure 2a shows the performance of an agent with hand coded object selection knowledge and Figure 2b, of an agent with learned object selection knowledge; the agents learn to play at the particular level and on average earn more reward than the memory agent, as our design allows more exploration of the environment. The average reward earned in both cases is close to the upper bound on the reward that can be earned in the particular instance, however the agent learns considerably faster when it learns an object selection strategy from its experience. This is because a human can program only very general object selection strategies, such as for a choice between a *coin* and a *Goomba*, select *Goomba*. However, if an agent is learning the selection knowledge itself, it can learn more specific selection strategies that involve relative distances. These strategies are very specific to the game instance, and hence converge faster.

Our framework also allows for inclusion of background knowledge in the design of agents which is beneficial in situations that are hard to learn through RL. Such knowledge can include spatial knowledge such as - *if an object x is on a platform p, and is-close(p, Mario), then is-close(x, Mario)*. The agent can learn to *climb* on the *platform* to reach the object. This leads to higher total reward (148.56 as compared to ˜145 in other designs) as shown in Figure 2c.

Limitations

When we used the proposed learning scheme at harder levels of the game, the agent failed to learn a good policy for play-

ing the game. We believe that it is because of the following reasons.

Structural Credit Assignment Problem

Observations from the harder levels of the game indicate the problem of credit assignment. The environment provides the agent with one composite reward signal for the complete state of the agent. However, in a complex environment there can be multiple sources of rewards. In Infinite Mario, the agent is rewarded when it achieves an object-related goal and that reward should be used to update the value function related to the specific object only. This distinction was lacking in the formulations described earlier. If the agent successfully collects a *coin*, while executing *tackle-monster* for a particular *monster m*, the positive reward so acquired is used to update the policy related to *tackle-monster*. The composite reward leads to spurious updates in policy.

The correct object-reward association could lead to better learning of object-related policies. In order to test this theory, we added a simple modification, and instead of using the environment supplied reward, we used a goal-directed reward while learning object-related MDPs. For example, while executing *tackle enemies* the agent would get positive reward only when it succeeded in *killing* the corresponding *enemies*. Figure 2d shows that adding this modification leads to faster learning. The performance at higher levels did not get better, and it could be because of more complex dynamics occurring in higher levels leading to a harder action selection problem. This issue is a candidate for further exploration.

Partial Observability

As noted in the previous section, to constrain the learning problem to one object at a time, we assumed that when the agent is executing a particular *behavior*, other objects do not affect the agent's selection strategy. However, such isolation is rare. In a general case, when an agent is executing a *behavior* for a particular object, other spatially close objects also affect the *behavior* and should be accounted for. In specific cases where the object density around Mario is high (see Figure 1b), this leads to problems in learning a good policy because of incorrect updates to the value function.

Conclusions and Future Work

Through this work, we have attempted to characterize the reinforcement learning problem designed around the game of Infinite Mario. The game domain is object-oriented, continuous, high-dimensional and has a high degree of relative motion between objects. These properties of the domain present some interesting challenges to an RL agent. An object-oriented approach can lead to tractable learning in complex environments such as this one. We proposed that a learning problem can be broken down into two inter-related problems, *learning object-related behaviors* and *action selection given a set of behaviors and objects*. We proposed an action selection policy based on SMDP learning, and presented encouraging results in specific instances of the game. The isolation assumption leads to partial observability in most general cases, leading to inefficient policies. We speculate that in a complex, multi-object environments there could be multiple sources of reward, and knowledge of the object-reward association could aid in learning better RL policies.

In the future, we will attempt to study the problem of multiple sources of rewards in a complex environment in detail. We are extremely interested in architectures (Shelton, 2001) that allow an agent to maintain and learn multiple, distinct object-related policies while acting in the environment and would like to investigate if these policies can be combined in simple ways to produce a good composite policy.

References

Barto, A. G., and Mahadevan, S. 2003. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems* 13(4):341–379.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research* 13(1):227–303.

Diuk, C.; Cohen, A.; and Littman, M. 2008. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine learning*, 240–247.

Driessens, K. 2001. Relational reinforcement learning. *Multi-Agent Systems and Applications* 271–280.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing plans to new environments in relational MDPs. In *Proceedings of International Joint Conference on Artificial Intelligence*.

John, B., and Vera, A. 1992. A GOMS analysis of a graphic machine-paced, highly interactive task. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 251–258.

Laird, J., and van Lent, M. 2001. Human-level AI's killer application: Interactive Computer Games. *AI Magazine* 22(2).

Marthi, B.; Russell, S.; Latham, D.; and Guestrin, C. 2005. Concurrent hierarchical reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, 1652.

Ponsen, M.; Spronck, P.; and Tuyls, K. 2006. Hierarchical reinforcement learning with deictic representation in a computer game. In *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, 251–258.

Rummery, G., and Niranjan, M. 1994. On-line Q-learning using Connectionist Systems. In *Tech. Report CUED/F-INFENG/TR166*. Cambridge University.

Shelton, C. R. 2001. Balancing Multiple Sources of Reward in Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 1082–1088.

Spronck, P.; Ponsen, M.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2006. Adaptive game AI with dynamic scripting. *Machine Learning* 63(3):217–248.

Sutton, R., and Barto, A. 1998. *Introduction to Reinforcement Learning*. MIT Press, 1st edition.

Tanner, B., and White, A. 2009. RL-Glue: Language-independent software for reinforcement-learning experiments. *The Journal of Machine Learning Research* 10:2133–2136.

Wintermute, S. 2010. Using Imagery to Simplify Perceptual Abstraction in Reinforcement Learning Agents. In *In Proceedings of the the Twenty-Fourth AAAI Conference on Artificial Intelligence*.