Towards Relational Hierarchical Reinforcement Learning in Computer Games

Marc Ponsen Pieter Spronck Karl Tuyls

MICC-IKAT, Universiteit Maastricht, PO Box 616, 6200 MD Maastricht, The Netherlands

Abstract

Computer games are challenging test beds for machine learning research. Without applying abstraction and generalization techniques, many traditional machine learning techniques, such as reinforcement learning, will fail to learn efficiently. In this paper we examine extensions of reinforcement learning that scale to the complexity of computer games. In particular we look at hierarchical reinforcement learning applied to a learning task in a real time-strategy computer game. Moreover, we provide a first step towards relational reinforcement learning in computer games by introducing a relational representation of the state features and actions. We found that hierarchical reinforcement learning significantly outperforms flat reinforcement learning for our task. We also show that a relational state representation allows the adaptive agent to learn a generalized policy, i.e., it is capable of transferring knowledge to unseen task instances.

1 Introduction

In reinforcement learning (RL) problems, an adaptive agent interacts with its environment and iteratively learns a policy. Policies are usually represented in a tabular format, where each cell includes a state or state-action value representing, respectively, the desirability of being in a state or the desirability of choosing an action in a state. In previous research, this approach has proven to be feasible in 'toy' domains with limited state and action spaces. In contrast, in more complex domains the number of states grows exponentially, resulting in an intractable learning problem. Modern computer games are typical examples of such complex domains. They present realistic models of real-world environments and offer a unique set of artificial intelligence (AI) challenges, such as dealing with huge state and action spaces and real-time decision making in stochastic and partially observable worlds.

Reducing the policy space through abstraction or applying generalization techniques is essential to enable efficient RL in computer games. In this paper we investigate how Q-learning can be hierarchically applied in a real time-strategy computer game and evaluate its convergence speed with respect to a flat setting. Hierarchical methods decompose a problem into a set of smaller problems that are then solved independently. We empirically validate the effectiveness of flat and

hierarchical RL in a sub-domain of the real-time strategy game Battle of Survival. Furthermore, we take the first step towards relational reinforcement learning by representing the learning task in a relational format. This allows the adaptive agent to transfer knowledge to unseen task instances.

The remainder of the paper is organized as follows. In Section 2 we will discuss related work. Section 3 describes reinforcement learning. Section 4 introduces hierarchical reinforcement learning and the algorithm used in the experiments. In Section 5 we will introduce the task we propose to solve. Section 6 describes a flat and hierarchical representation of this problem, while Section 7 presents experimental settings and compares results. We conclude in Section 8.

2 Related Work

For an extensive overview of RL related work we refer to [13, 8]. Several Hierarchical RL (HRL) techniques have been developed e.g., MAXQ [3], Options [13], HAMs [10], HASSLE [1] and HQ-learning [15]. A summary of work on HRL is given by [2]. Relational RL is proposed in [6] and further examined in [5]. In this section we specifically focus on RL research in computer games. Previous RL research in computer games either assumed appropriate abstractions and generalizations or addressed only very limited computer game scenarios. For example, Spronck et al. [12] and Ponsen et al. [11] implemented a RL inspired technique called *dynamic scripting* in several computer games. They report good learning performances on a challenging task: learning to win computer games. However, priors were considerably reduced state and action spaces (often temporally extended actions). In contrast, in this research we allow agents to plan with actions that take exactly one time step.

Driessens [5] combined RL with regression algorithms to generalize in the policy space. He evaluated his relational RL approach in two computer games. Similarly, we employ a relational format for the state and actions features. Guestrin et al. [7] also learned generalized policies in a limited real-time strategy game domain by solving a relational Markov decision process.

Marthi et al. [9] applied hierarchical RL to scale to complex environments. They learned navigational policies for agents in a limited real-time strategy computer game domain. Their action space consisted of partial programs, essentially high-level pre-programmed behaviors with a number of *choice points* that were learned using Q-learning. Our work differs from theirs in that we use a different hierarchical RL technique.

3 Reinforcement Learning

Most RL research is based on the framework of Markov decision processes (MDPs). MDPs are sequential decision making problems for fully observable worlds with a Markovian transition model. MDPs can be defined by a tuple $(s_0, t, S, A, \delta, r)$. Starting in an initial state s_0 at each discrete time-step $t = 0, 1, 2, \ldots$ an adaptive agent observes an environment state s_t contained in a finite set of states $S =$

$\{s_1, s_2, \ldots, s_n\}$, and executes an action a from a finite set $A = \{a_1, a_2, \ldots, a_m\}$ of admissible actions. The agent receives an immediate reward $r : S \times A \rightarrow \mathbb{R}$, and moves to a new state s' depending on a (unknown) transition probability $\delta : S \times A \rightarrow S$. The learning task in MDPs is to find a policy $\pi : S \rightarrow A$ for selecting actions that maximizes a value function $V^{\pi}(s_t)$ for all $s_t \in S$.

Temporal difference learning methods, and in particular Q-learning [14], are popular to solve MDPs because these require no model (i.e., transition and reward functions are unknown) and can be applied online. These characteristics make Q-learning a suitable learning algorithm for computer games. Namely, obtaining correct and complete models, even for limited computer game scenarios, can be difficult. Furthermore, learning in computer games should preferably take place online. The *one-step* Q-learning update rule is denoted as:

$$Q(a, s) \rightarrow (1 - \alpha)Q(a, s) + \alpha \left[r + \gamma \max_{a'} Q(a', s') \right] \quad (1)$$

where α is the step-size parameter, and γ the discount-rate.

4 Hierarchical Reinforcement Learning

Hierarchical RL (HRL) is an intuitive and promising approach to scale up RL to more complex problems. In HRL, a complex task is decomposed into a set of simpler subtasks that can be solved independently. Each subtask in the hierarchy is modeled as a single MDP and allows appropriate state, action and reward abstractions to augment learning compared to a flat representation of the problem. Additionally, learning in a hierarchical setting can facilitate generalization, e.g., knowledge learned by a subtask can be transferred to other subtasks.

HRL relies on the theory of Semi-Markov decision processes (SMDPs). SMDPs differ from MDPs in that actions in SMDPs can last multiple time steps. Therefore, in SMDPs actions can either be primitive actions (taking exactly 1 time-step) or temporally extended actions. While the idea of applying HRL in complex domains such as computer games is appealing, few studies in this respect actually exist [2].

We adopted a HRL method similar to Hierarchical Semi-Markov Q-learning (HSMQ) described in [4]. HSMQ learns policies simultaneously for all non-primitive subtasks in the hierarchy. Each subtask will learn its own $Q(p, s, a)$ function, which is the expected total reward of performing task p starting in state s, executing action a and then following the optimal policy thereafter. Subtasks in HSMQ include termination predicates. These partition the state space S into a set of active states and terminal states. Subtasks can only be invoked in active states, and subtasks terminate when the state transitions from an active to a terminal state. Subtasks may include pseudo-reward functions [3], which specify a pseudo-reward for each transition from an active state to a terminal state. The pseudo-rewards tell how desirable each of the terminal states are for this subtask. Algorithm 1 outlines our HSMQ inspired algorithm. Q-values for primitive subtasks are updated with the one-step Q-learning update rule, while the Q-values for non-primitive subtasks are updated based on the reward r collected during execution of the subtask and a pseudo reward $\hat{R}$.

Algorithm 1: HSMQ Algorithm

```
1  Function HSMQ(state s,subtask p) returns float;
2  Let Totalreward = 0;
3  while (p is not terminated) do
4      Choose action a = Π(s);
5      if a is primitive then
6          Execute a, observe one-step reward r and result state s';
7      else if a is non-primitive subtask then
8          r := HSMQ(s, a) , which invokes subtask a and returns the total
           reward received while a executed
9      Totalreward = Totalreward + r;
10     if a is primitive then
11         Q(a, s) → (1 − α)Q(a, s) + α [r + γ max_{a'} Q(a', s')];
12     else if a is non-primitive subtask then
13         Q(a, s) → (1 − α)Q(a, s) + α [r + R̂];
14 end
15 return Totalreward;
```

5 Reactive Navigation Task

Real-Time Strategy (RTS) games require players to control a civilization and use military force to defeat all opposing civilizations that are situated in a virtual battlefield in real time. In this study we focus on a single learning task in RTS games: we learn a policy for a worker unit in the Battle of Survival (BoS) game. BoS is a RTS game created with the open-source engine Stratagus. A worker unit should be capable of effective navigation and avoiding enemies. We captured these tasks in a simplified BoS scenario. This scenario takes place in a fully observable world that is 32 by 32 grid cells large and includes two units: a worker unit (the adaptive agent) and an enemy soldier. The adaptive agent has to move to a certain goal location. Once the agent reaches its goal, a new random goal is set. The enemy soldier randomly patrols the map and will shoot at the worker if it is in firing range. The scenario continues for a fixed time period or until the worker is destroyed by the enemy soldier.

Relevant properties for our task are the locations of the worker, soldier and goal. All three objects can be positioned in any of the 1024 different locations. A propositional format of the state space describes each state as a feature vector with attributes for each possible property of the environment, which amounts to 2^{30} different states. As such, a tabular representation of the value functions is too large to be feasible. A relational feature representation of the state space identifies objects in the environment and defines properties for these objects and relations between them [5]. This reduces state space complexity and facilitates generalization. We will discuss the relational state features used for this task in Section 6.

The proposed task is complex for several reasons. First, the state space without any abstractions is enormous. Second, the game state is also modified by an enemy unit, whose random patrol behaviour complicates learning. Furthermore, each new task instance is generated randomly (i.e., random goal and enemy patrol behavior), so the worker has to learn a policy that generalizes over unseen task instances.

6 Solving the Reactive Navigation Task

We compare two different ways to solve the reactive navigation task, namely using flat RL and HRL. For a **flat representation** of our task, the state representation can be defined as the Cartesian-product of the following four relational features: `Distance(enemy,s)`, `Distance(goal,s)`, `DirectionTo(enemy,s)` and `DirectionTo (goal,s)`. The function `Distance` returns a number between 1 and 8 or a string indicating that the object is more than 8 steps away in state s, while `DirectionTo` returns the relative direction to a given object in state s. Using 8 possible values for the `DirectionTo` function, namely the eight directions available on a compass rose, and 9 possible values for the `Distance` function, the total state space is drastically reduced from 2^{30} to a mere 5184 states. The size of the action space is 8, namely containing actions for moving in each of the eight compass directions. The scalar reward signal r in the flat representation should reflect the relative success of achieving the two concurrent sub-goals (i.e., moving towards the goal while avoiding the enemy). The environment returns a +10 reward whenever the agent is located on a goal location. In contrast, a negative reward of -10 is returned when the agent is being fired at by the enemy unit. Each primitive action always receives the usual reward of -1. An immediate concern is that both sub-goals are often in competition. Certainly we can consider situations where different actions are optimal for the two sub-goals, although the agent can only take one action at a time. An apparent solution to handle these two concurrent sub-goals is applying a hierarchical representation, which we discuss next.

In the **hierarchical representation**, illustrated in Figure 1, the original task is decomposed into two simpler subtasks that solve a single sub-goal independently. The *to goal* subtask is responsible for navigation to goal locations. Its state space includes the `Distance(goal,s)` and `DirectionTo(goal,s)` relational features. The *from enemy* subtask is responsible for evading the enemy unit. Its state space includes the `Distance(enemy,s)` and `DirectionTo(enemy,s)` relational features. The action spaces for both subtasks include the primitive actions for moving in all compass directions. The two subtasks are hierarchically combined in a higher-level *navigate* task. The state space of this task is represented by the new `InRange(goal,s)` and `InRange(enemy,s)` relational features, and its action space consists of the two subtasks that can be invoked as if they were primitive actions. `InRange` is a function that returns *true* if the distance to an object is 8 or less in state s, and *false* otherwise. The *to goal* and *from enemy* subtasks terminate at each state change on the root level, e.g., when the enemy (or goal) transitions from in range to out of range and vice versa. The pseudo-rewards for both subtasks were set to +100 whenever the subtask succeeded in achieving its

task while it was active and 0 otherwise. The *navigate* task never terminates while the primitive subtasks always terminate after execution. The state spaces for the two subtasks are of size 72, and for *navigate* of size 4. Therefore, the state space complexity in the hierarchical representation is approximately 35 times less than with the flat representation presented previously. Additionally, in the hierarchical setting we are able to split the reward signal, one for each subtask, so they do not interfere. The *to goal* subtask rewards solely moving to the goal (i.e., only process the +10 reward when reaching a goal location). Similarly, the *from enemy* subtask only rewards evading the enemy. Based on these two reward signals and the pseudo-rewards, the root *navigate* task is responsible for choosing the most appropriate subtask. For example, lets assume the worker at a certain time decided to move to the goal and it took the agent 7 steps to reach it. The reward collected while the *to goal* subtask was active is -7 (reward of -1 for all primitive actions) and +10 (for reaching the goal location) resulting in a +3 total reward. This reward is used to update the Q-values for the *to goal* subtask. Additionally, a pseudo-reward of +100 is received because *to goal* successfully terminated, resulting in a total reward of +103 that is propagated to the *navigate* subtask, that is used to update its Q-values.

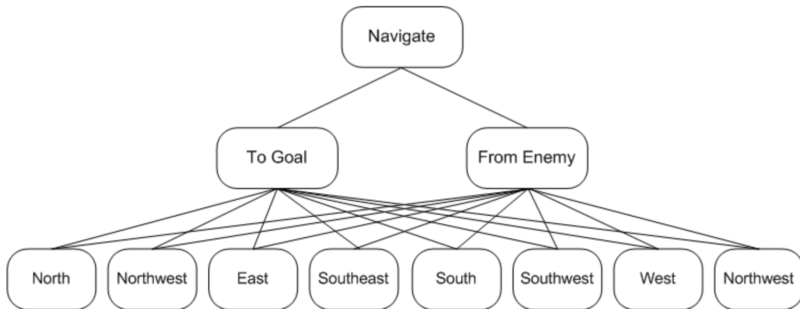

Figure 1: Hierarchical decomposition of the reactive navigation task.

7 Experimental Results

We evaluated the performance of flat RL and HRL in the reactive navigation task. The step-size and discount-rate parameters were set to respectively 0.2 and 0.7. These values were determined during initial experiments. We emphasized longer exploration for the *to goal* and *from enemy* compared to the *navigate* task, since more Q-values require learning for these subtasks. Therefore, we used Boltzmann action selection with a relatively high (but decaying) temperature for the subtasks and ϵ-greedy action selection at the top level, with ϵ set to 0.1 [13].

A training session (when Q-values are adapted) lasted for 30 episodes. An episode terminated when the adaptive agent was destroyed or until a fixed time limit was reached. During training, random instances of the task were generated, i.e., random initial starting locations for the units, random goals and random enemy patrol behaviour. After a training session, we empirically validated the

current policy on a test set consisting of 5 fixed task instances that were used throughout the entire experiment. These included fixed starting locations for all objects, fixed goals and fixed enemy patrol behaviour. We measured the performance of the policy by counting the number of goals achieved by the adaptive agent (i.e., the number of times the agent was successful at reaching the goal location before it was destroyed or time ran out) by evaluating the greedy policy. We ended the experiment after 1500 training episodes. The experiment was repeated 5 times and the averaged results are shown in Figure 2.

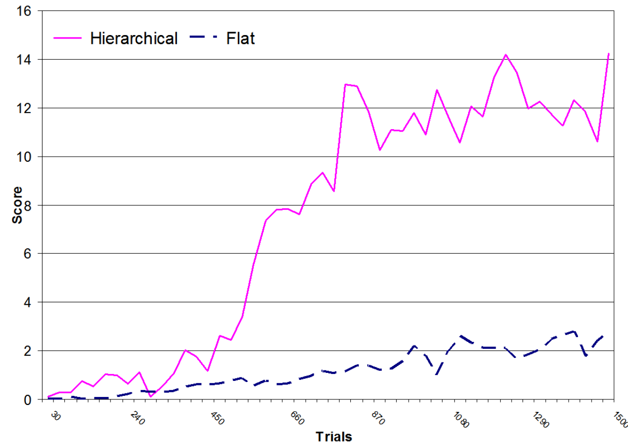

Figure 2: The average performance of Q-learning over 5 experiments in the reactive navigation task for both flat and HRL. The x-axis denotes the number of training trials and the y-axis denotes the average number of goals achieved by the agent for the tasks in the test set.

From this figure we can conclude that HRL clearly outperforms flat RL: we see faster convergence to a suboptimal policy for HRL, while flat RL is still in the early stages of learning. HRL allowed more state abstractions, thus resulting in fewer Q-values that required learning. Furthermore, HRL is more suitable in dealing with concurrent and competing subtasks due to the split (and therefore more informative) reward signal. We expect that even after considerable learning with flat RL, HRL will still achieve a higher overall performance.

8 Conclusion

We investigated the application of RL to a challenging and complex learning task, namely navigation of a worker unit in an RTS game. We discovered that a relational feature representation of the state-action space allows an adequate reduction of the number of states to make the learning task feasible. Additionally, a generalized policy was learned that scales to new task instances. We further found that by imposing a hierarchy on the problem, its complexity was further reduced and

produced significantly better results in all aspects than a flat RL algorithm.

For future work we plan to extend existing and investigate new abstraction and generalization algorithms for RL. We will work towards a fully relational RL algorithm. Currently, we adopted a relational feature representation but the state space is still represented propositionally. Also, the Q-values are currently represented in a lookup table. We plan to investigate regression techniques to build a Q-value generalization.

References

[1] B. Bakker and J. Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the Conference on Intelligent Autonomous Systems (IAS-08)*, pages 438–445, 2004.

[2] A.G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Application*, 13(4):341–379, 2003.

[3] T.G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[4] T.G. Dietterich. An overview of MAXQ hierarchical reinforcement learning. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA-00), Lecture Notes in Computer Science*, volume 1864, pages 26–44, 2000.

[5] K. Driessens. *Relational Reinforcement Learning*. PhD thesis, Katholieke Unversiteit Leuven, 2004.

[6] S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.

[7] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.

[8] L.P. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[9] B. Marthi, S. Russell, and D. Latham. Writing Stratagus-playing agents in concurrent ALisp. 2005.

[10] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of Conference on Advances in Neural Information Processing Systems (NIPS-97)*, pages 1043–1049, 1997.

[11] M. Ponsen and P. Spronck. Improving adaptive game AI with evolutionary learning. In *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, pages 389–396, 2004.

[12] P. Spronck, M. Ponsen, E. Postma, and I.G. Sprinkhuizen-Kuyper. Adaptive game AI with dynamic scripting. *Machine Learning: Special Issue on Computer Games*, 2006.

[13] R. Sutton and A. Barto. *Reinforcement Learning: an introduction*. MIT Press, Cambridge, MA, USA, 1998.

[14] C.J.C.H. Watkins. *Learning with Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[15] M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.