Hierarchical Reinforcement Learning in Computer Games

Marc Ponsen, Pieter Spronck, Karl Tuyls

Maastricht University / MICC-IKAT.
{m.ponsen,p.spronck,k.tuyls}@cs.unimaas.nl

Abstract. Hierarchical reinforcement learning is an increasingly popular research field. In hierarchical reinforcement learning the complete learning task is decomposed into smaller subtasks that are combined in a hierarchical network. The subtasks can then be learned independently. A hierarchical decomposition can potentially facilitate state abstractions (i.e., bring forth a reduction in state space complexity) and generalization (i.e., knowledge learned by a subtask can be transferred to other subtasks). In this paper we empirically evaluate the performance of two reinforcement learning algorithms, namely Q-learning and dynamic scripting, in both a flat (i.e., without task decomposition) and a hierarchical setting. Moreover, this paper provides a first step towards relational reinforcement learning by introducing a relational representation of the state features and actions. The learning task in this paper involves learning a generalized policy for a worker unit in a real time-strategy game called BATTLE OF SURVIVAL. We found that hierarchical reinforcement learning significantly outperforms flat reinforcement learning for our task.

1 Introduction

In reinforcement learning (RL) problems, an adaptive agent interacts with its environment and iteratively learns a *policy*, i.e., it learns *what* to do *when* in order to achieve a certain goal (Sutton & Barto, 1998; Kaebling, 1996). 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 action and state 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, real-time decision making in stochastic and partially observable worlds, and learning player models.

Reducing the policy space through abstraction or applying generalization techniques is essential to enable efficient RL in computer games. In this paper we will evaluate whether learning augments with *dynamic scripting* (Spronck *et al*., 2006), a greedy RL method, and with *Q-learning* (Watkins, 1989), when applying it in a *hierarchical* 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.

The remainder of the paper is organized as follows. In Section 2, we will discuss related work. In Section 3, we will introduce the task we propose to solve. Section 4 describes the two learning algorithms that were empirically evaluated in this paper. Section 5 introduces hierarchical reinforcement learning and describes the hierarchical Semi-Markov learning algorithm used in the experiments. Section 6 describes a flat and hierarchical representation of the problem presented in Section 3, while Section 7 presents experimental settings and compares the experimental results achieved with flat and hierarchical RL for both learning algorithms. We discuss these results and conclude in Sections 8 and 9.

2 Related Work

For an extensive overview of RL related work we refer to Kaebling (1996), Sutton & Barto (1998) and Hoen *et al.* (2006). Hierarchical RL related work is described in Barto and Mahedevan (2003). In this Section we specifically focus on RL related 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.* (2006) implemented a RL inspired technique called *dynamic scripting* in a computer-role playing game called NEVERWINTER NIGHTS™. They report good learning performances: adaptive agents were able to rapidly optimize their combat behavior against a set of computer opponents. Similarly, Ponsen *et al.* (2004) implemented the dynamic scripting technique in a real-time strategy (RTS) game called WARGUS. They addressed the problem of learning to win complete games. Again, the learning performances were encouraging. However, priors for both implementations of dynamic scripting were considerably reduced state and action spaces and the availability of high-quality domain knowledge. For instance, in WARGUS only 20 states were distinguished and each state on average included 30 admissible temporally extended (i.e., high-level) actions. Additionally, actions were assumed to be mostly of high-quality, i.e., produce acceptable game behavior. In both experiments actions were manually designed. However, Ponsen *et al.* (2005) showed that it also possible to generate high-quality actions using a genetic algorithm. In contrast, in this research we allow agents to plan with primitive actions.

Graepel *et al.* (2004) used the SARSA algorithm to learn state-action values for an agent in a fighting game called TOA FENG™. Besides using an abstracted representation of the state and action space, they generalized over the state-action values with different function approximators. Similarly, Driessens (2004) combined RL with relational regression algorithms to generalize in the policy space. He evaluated his relational RL approach (RRL) in two computer games. Similar to Driessens (2004), we employ a relational format for the actions and state features. Using this representation, we are able to learn a generalized policy, i.e., one that scales to unseen task instances. Guestrin *et al.* (2003) also learned generalized policies in a limited RTS domain by solving a relational MDP (RMDP).

Hartley *et al.* (2004) use value iteration to solve a grid-world navigation task. Their algorithm found an optimal policy in a relatively small 10 by 10 grid world. Our learning task also focuses on navigation but is significantly more complex, and finding optimal policies for our task is probably not feasible without appropriate abstractions and generalizations.

Marthi *et al.* (2005) applied hierarchical RL to scale to complex environments. They learned navigational policies for agents in a limited WARGUS 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 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, namely, 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 typical task for worker units is resource gathering. A worker 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 cells large and includes two units: a worker unit (the adaptive agent) and an enemy soldier. The adaptive agent is given orders 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 using a propositional format of the state space 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 (Driessens, 2004). This reduces state space complexity and facilitates generalization.

This 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 location and enemy patrol behaviour), so the worker has to learn a policy that generalizes over unseen task instances.

Fig. 1. Screenshot of the reactive navigation task in the BoS game. In this example, the adaptive agent is situated at the bottom. Its task is to move to a goal position (the dark spot right to the center) and avoid the enemy soldier (situated in the upper left corner) that is randomly patrolling the map.

4 Reinforcement Learning

Most RL research is based on the framework of Markov decision processes (MDP). 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 $t = 0,1,2,3\ldots$ an adaptive agent observes an environment state S_t contained in a finite set of states $S = \{S_1,S_2,S_3,\ldots,S_n\}$, and executes an action a from a finite set $A = \{A_1,A_2,A_3,\ldots,A_m\}$ of admissible actions. The agent receives an immediate reward $r: S \times A \rightarrow \Re$, 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 $\pi^*: S \times A$ for selecting actions that maximizes a value function $V^{\pi}(S_t)$ for all $S_t \in S$.

4.1 Reinforcement Learning with Q-Learning

Several approaches exist for learning optimal policies, such as dynamic programming, Monte Carlo methods and temporal-difference learning methods (Sutton & Barto, 1998). Temporal difference (TD) learning methods, and in particular Q-learning (Watkins 1989), are popular because these require no model (i.e., transition and reward functions can be unknown) and can be applied online (Sutton & Barto,

1998). Obtaining correct models, even for limited computer game scenarios, can be difficult or even impossible. Even though our learning task as described in Section 3 is fully observable and the effects of actions are deterministic, to be able to scale up to complete RTS games we will assume that transition functions are unknown. Since learning in computer games should preferably take place online, we choose to implement *one-step* Q-learning to solve our learning task. The *one-step* Q-learning update rule is denoted as:

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

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

4.2 Reinforcement Learning with Dynamic Scripting

Dynamic scripting (Spronck *et al.*, 2006) is a RL inspired machine-learning technique designed for creating adaptive computer game agents. It employs on-policy value iteration to optimize state-action values based solely on a scalar reward signal. Consequently, it is only concerned with maximizing immediate reward. Action-selection is implemented with a *softmax* method (Sutton and Barto, 1998). The dynamic scripting learning update rule is denoted as:

$$Q(s,a) \rightarrow Q(s,a) + [r] \quad 2$$

The reward *r* in the dynamic scripting framework is typically designed with prior knowledge of *how* to achieve a certain goal and cause high discrepancies in the state-action values (Spronck *et al.*, 2006; Ponsen *et al.*, 2004). Consequently, this will lead to faster exploitation, i.e., the chance that the *greedy* action is selected increases.

Dynamic scripting is designed for online use in computer games and therefore adaptive agents start exploiting knowledge in a few trials only. However, the technique also allows balancing exploitation and exploration by maintaining a minimum and maximum selection probability for all actions. Elementary solution methods such as TD or Monte-Carlo learning update state-action values only after they are executed (Sutton & Barto, 1998). In contrast, dynamic scripting updates all state-action values in a specific state through a redistribution process (Spronck *et al.*, 2006). For example, if a certain action receives a large reward in a specific state, other actions in this state receive penalties that when summed up equal the amount of the reward.

We should note that because of these properties, dynamic scripting cannot guarantee convergence. This actually is essential for its successful use in computer games. The learning task in a computer game constantly changes (e.g., an opponent player may choose to switch tactics) and aiming for an optimal policy may result in overfitting to a specific task. Dynamic scripting should be capable of generating and dealing with a variety of behaviors, and should respond quickly to changing game dynamics. Additionally, we would argue that suboptimal policies are acceptable in computer games. Our dynamic scripting implementation is denoted as:

```
Initialize Q(s,a) to arbitrary value for all s, a
Repeat (for each episode)
Initialize S0
  Repeat (for each step of the episode)
      Choose action a = π(s), Softmax exploration
      (Boltzmann)
      Execute action a, observe r and s`
      Q(s,a)→ Q(s,a) + [r]
      Distribute (-[r]) over all admissible Q(s,a)
      s→s`
  Until (s is terminal)
End
```

5 Hierarchical Reinforcement Learning

Hierarchical RL (HRL) is an intuitive and promising approach to scaling up RL to more complex problems. In HRL, a complex task (e.g., winning a RTS game) is decomposed into a set of simpler subtasks (e.g., resource gathering in RTS games) that can be solved independently. Each subtask in the hierarchy is modelled as a single MDP and allows appropriate state, action and reward abstractions so as 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 others.

HRL relies on the theory of Semi-Markov decision processes (SMDP). 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. Several HRL techniques have been designed, most notably MAXQ (Dietterich, 2000a), Options (Sutton *et al.*, 1998) and HAMs (Parr *et al.*, 2001). Barto and Mahedevan (2003) provide an overview of these and other HRL techniques. Currently, for most HRL approaches the task hierarchy and subtask abstractions are designed manually (Barto and Mahedevan, 2003), although Hengst (2002) showed that the HEXQ technique automatically learned a MAXQ task hierarchy for Dietterich's taxi problem (Dietterich, 2000a). While the idea of applying HRL methods in complex domains such as computer games is appealing, very few studies in this respect actually exist (Barto and Mahedevan, 2003).

We adopted a HRL method described in Dietterich (2000b), namely Hierarchical Semi-Markov Learning (HSML). HSML learns policies simultaneously for all non-primitive subtasks in the hierarchy. Each subtasks 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 HSML 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. The HSML algorithm is denoted as:

```
Function HSML(state s,subtask p) returns float
    Let Totalreward = 0
    While (p is not terminated) do
          Choose action a = π(s), Softmax exploration
          (Boltzmann)
          Execute a
          if a is primitive
                Observe one-step reward r
          else
                r := HSML(s,a), which invokes subrou-
                tine a and returns the total reward re-
                ceived while a executed.
          Totalreward = Totalreward + r
          s→s`
          Update Q(p,s,a), e.g., with Q-learning update
    End
    Return Totalreward
End
```

6 Solving the Reactive Navigation Task

We compare two different ways to solve the reactive navigation task, namely using *flat* and *hierarchical* RL. 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), DirectionTo(enemy), Distance(goal) and DirectionTo(goal).* The function *Distance* returns a number between 1 and 8 or a string indicating that the object is more than 8 steps away, while *DirectionTo* returns the relative direction to a given object. 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 to a mere 5184 states. This number is significantly less than the size of the state space without abstractions. The size of the action space is 8, namely containing actions for moving in each of the eight compass directions. This results in a total of 8 * 5184 = 41472 Q-values that require learning.

The scalar reward signal *r* in the flat representation should reflect the relative success of achieving the two concurrent subtasks. With Q-learning, the environment returns a positive reward of +10 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. The scalar reward used for dynamic scripting is more knowledge intensive and is denoted as:

$$r : S \times A \rightarrow (\text{Distance(enemy, } s\text{`}) - \text{Distance(enemy, } s)) + (\text{Distance(goal, } s) - \text{Distance(goal, } s\text{`})) \quad (3)$$

The first term rewards moving away from the enemy, while the second term rewards moving closer to the goal. The knowledge lies in the fact that this reward tells the learning agent *how* to achieve the goal (e.g., to evade enemy move as far away from it as possible).

An immediate concern is that both sub-goals (moving towards a goal and avoiding the enemy) are often in competition and that designing a suitable reward is difficult.

Certainly we can consider situations where different actions are optimal for the two sub-goals. Driessens (2004) was confronted with the same problem in his Digger task. An apparent solution to handle these two concurrent sub-goals is applying a hierarchical representation, which we discuss next.

A hierarchical representation of the task is illustrated in Figure 2. The original task is decomposed into two simpler subtasks that solve a single sub-goal independently. The *´to goal´* task is responsible for navigation to goal locations. Its state space includes the *Distance(goal)* and *DirectionTo(goal)* relational features. The *´from enemy´* task is responsible for evading the enemy unit. Its state space includes the *Distance(enemy)* and *DirectionTo(enemy)* 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)* and *InRange(enemy)* 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, 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 *'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 the *´navigate´* of size 4. We employed a fixed policy for the *'navigate'* root task, i.e., learning only took place at the two non-primitive subtasks (the *´navigate´* policy favours the *´from enemy´* subtask when the enemy is in range and the *´to goal´* subtask otherwise). Therefore, in the hierarchical representation this leads to 72 * 8 * 2 = 1152 Q-values that require learning, which is significantly 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 (e.g., in the case of Q-learning only process the +10 reward when reaching a goal location). Similarly, the *´from enemy´* subtask only rewards avoiding the enemy.

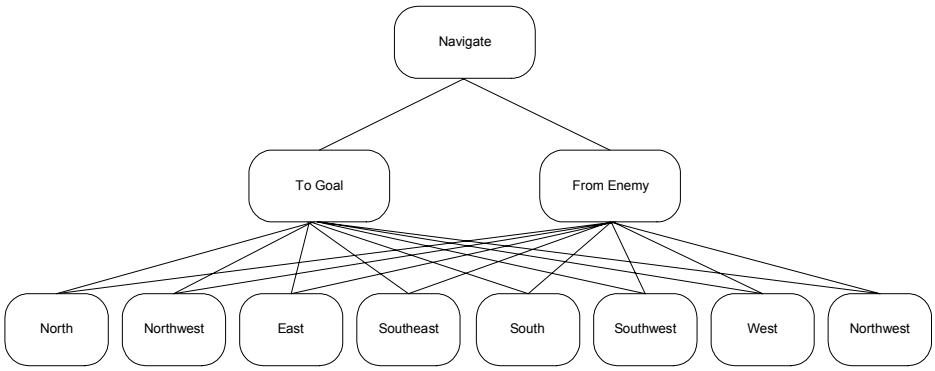

Fig. 2. Hierarchical decomposition of the reactive navigation task

7 Experimental Results

We evaluated the performance of flat RL and HRL in the reactive navigation task. During training, state-action values were adapted using the Q-learning and dynamic scripting learning algorithms. The step-size and discount-rate parameters for Q-learning were set to 0.7. These values were determined during initial experiments. A training session for Q-learning lasted for 30 episodes and 10 episodes for dynamic scripting (because of the knowledge intensive reward signal we expected dynamic scripting to converge faster). 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 learned by Q-learning and dynamic scripting on a separately generated 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 (i.e., the number of times the agent was successful at reaching the goal location before it was destroyed or time ran out) achieved by the adaptive agent by evaluating the greedy policy. For Q-learning we ended the experiment after 1500 training episodes and for dynamic scripting after 500 training episodes. We repeated the experiments with both learning algorithms 5 times and the averaged results are shown in Figure 3 and 4.

Hierarchical RL clearly outperforms flat RL with both the Q-learning and dynamic scripting algorithm: we see faster convergence to a suboptimal policy and the overall performance increases. HRL allowed more state abstractions, thus resulting in fewer Q-values that required learning. Furthermore, HRL is more suitable in dealing with two concurrent and often competing subtasks.

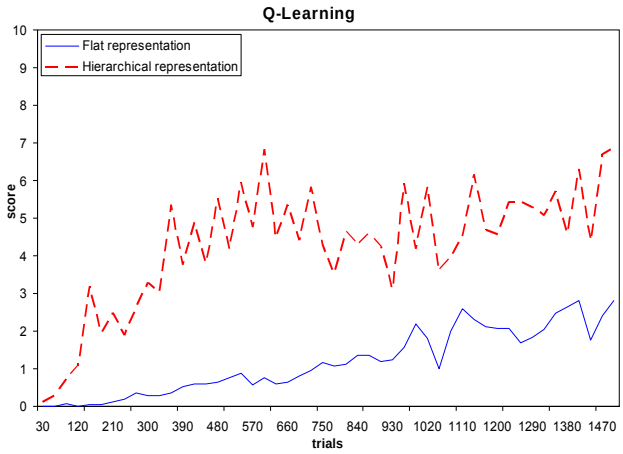

Fig. 3. The average performance of Q-learning over 5 experiments in the reactive navigation task for both flat and hierarchical RL. The x-axis denotes the number of training trials and the y-axis denotes the average number of goals achieved by the agent for the tasks in the test set.

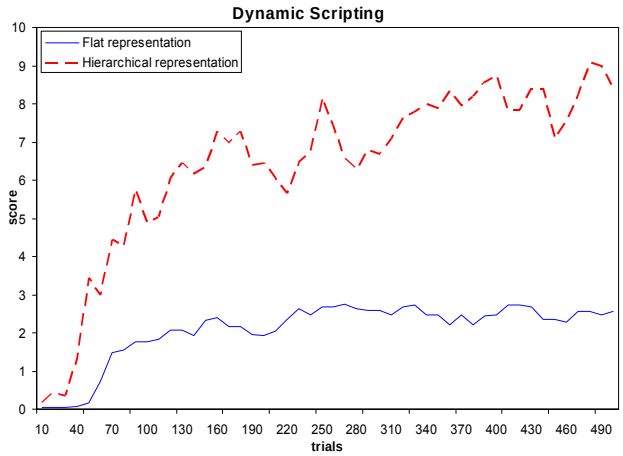

Fig. 4. The average performance of dynamic scripting over 5 experiments in the reactive navigation task for both flat and hierarchical RL. The x-axis denotes the number of training trials and the y-axis the number of goals achieved. Note that training time for dynamic scripting is three times shorter compared to Q-learning.

8 Discussion

For the experiments described in the previous Section we employed a fixed policy for the *´navigate´* task. The reason is that, when we ran experiments with learning on all non-primitive subtasks, we noticed that inferior policies were learned for the *´navigate´* subtask. We clarify this as follows. In the experiments, we employed a decaying step-size parameter, with the decay being determined by the number of updates a particular Q-value already underwent. The few Q-values for the *´navigate´* policy are updated extremely often and particularly in the early stages of learning the rewards received from child subtasks are extremely noisy. By the time that the *´to goal´* and *´from enemy´* subtasks have converged to reasonable policies and more informative reward signals are returned, learning for the '*navigate´* task has stopped because of a low step-size parameter. It is proved in theory that convergence can only be guaranteed with an *appropriately decaying* step-size parameter (Sutton & Barto, 1998), but practice shows that choosing this decay factor can be very challenging. We then decided to run experiments using a fixed step-size parameter. This resulted in a more sensible but still non-optimal policy for the *´navigate´* task, namely the policy learned to always favour walking towards the goal. We expect this can be subscribed to the design of our reward signals. A successful execution of the *´to goal´* subtask yields in a +10 reward. A successful execution of the *´from enemy´* task at best yields in a zero reward. Therefore, we expect that after considerable learning the *´to goal´* task will always be favoured, in all states. To resolve these problems for the present time, we decided to implement a fixed policy for the *´navigate´* task. However, in future work we will certainly investigate concurrent learning at different levels of the hierarchy, since the number of levels in the hierarchy will increase for more complex environments.

9 Conclusion

We researched the application of RL to a challenging and complex learning task, namely navigation of a worker unit in an RTS game. Learning in computer games is still very much in flux and recognized as an open and difficult problem in the learning agents research community. We discovered that a relational feature representation of the state-action space would allow 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 a hierarchical RL algorithm would produce 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. For example, the HSML algorithm can be extended to the MAXQ algorithm (Dietterich, 2000a) by introducing value decomposition. This will bring forth sharing in the representation of the value function and allows more forms of state abstraction (Dietterich, 2000a). Also 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-function generalization. Our ultimate goal is to scale up the learning algorithms to develop policies that successfully play complete RTS games.

Acknowledgements

The first author is sponsored by DARPA and managed by NRL under grant N00173-06-1-G005. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NRL, or the US Government.

The second author is funded by a grant from the Netherlands Organization for Scientific Research (NWO grant No 612.066.406).

The third author is sponsored by the Interactive Collaborative Information Systems (ICIS) project, supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024.

References

1. Barto, A., Mahadevan, S.: Recent Advances in Hierarchical Reinforcement Learning, Discrete Event Dynamic Systems: Theory and Application, 13(1-2):41-77, (2003)
2. Dietterich, T.: Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, JAIR, 13:227-303, (2000a)
3. Dietterich, T.: Overview of MAXQ Hierarchical Reinforcement Learning, In Proceedings of the Symposium on Abstraction, Reformulation and Approximation SARA, Lecture Notes in Artificial Intelligence (pp. 26-44), New York: Springer Verlag. Postscript preprint © Springer-Verlag, (2002b)
4. Driessens, K.: Relational Reinforcement Learning, Ph.D. thesis, Katholieke Unversiteit Leuven, (2004)

5. Graepel, T., Herbrich R., Gold J.: Learning to Fight, Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004) 193–200, University of Wolverhampton, (2004)
6. Guestrin, C., Koller, D., Gearhart, C., Kanodia, N.: Generalizing plans to new environments in relational MDPs, Proceedings of the Eighteenth International Joint Conference in Artificial Intelligence 1003-1010. Acapulco, Mexico: Morgan Kaufmann, (2003)
7. Hartley, T., Mehdi, Q., Gough, N.: Using Value Iteration to Solve Sequential Decision Problems in Games, Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004) 293–297, University of Wolverhampton, (2004)
8. Hengst, B.: Discovering Hierarchy in Reinforcement Learning with HEXQ, In Machine Learning: Proceedings of Nineteenth International Conference of Machine Learning, (2002)
9. 't Hoen, P.J., Tuyls, K.,Panait L., Luke S., La Poutre J.A.: An Overview of Cooperative and Competitive Multiagent Learning. In Learning and Adaptation in MAS, LNCS Volume 3898, pages 1-49, Springer-Verlag (2006).
10. Kaebling, L., Littman, M., Moore, A.: Reinforcement Learning: A survey. Journal of Artificial Intelligence, 4:237-285, (1996)
11. Marthi, B., Russell, S., Latham, D.: Writing Stratagus-playing Agents in Concurrent ALisp. Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. Edinburgh, Scotland, (2005)
12. Parr, R., Russell, S.: Reinforcement Learning with Hierarchies of Machines, Advances in Neural Information Processing Systems, 9, (1997)
13. Ponsen, M., Muñoz-Avila, H., Spronck P., Aha D.W.: Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning, Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence, Pittsburgh, PA: Morgan Kaufmann, (2005)
14. Ponsen, M., Spronck, P.: Improving adaptive game AI with evolutionary learning, Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004) 389-396, Reading, UK: University of Wolverhampton Press, (2004)
15. Spronck, P., Ponsen, M.: Sprinkhuizen-Kuyper, I., Postma. E, Adaptive Game AI, Journal of Machine Learning, Special issue on Computer Games, (2006)
16. Sutton, R., Barto, A.: Reinforcement Learning an introduction. The MIT Press, Cambridge, MA, (1998)
17. Watkins, C.J.C.H.: Learning with Delayed Rewards. PhD thesis, Cambridge University, (1989)