Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft:Broodwar

Stefan Wender, Ian Watson

***Abstract*—This paper presents an evaluation of the suitability of reinforcement learning (RL) algorithms to perform the task of micro-managing combat units in the commercial real-time strategy (RTS) game StarCraft:Broodwar (SC:BW). The applied techniques are variations of the common Q-learning and Sarsa algorithms, both simple one-step versions as well as more sophisticated versions that use eligibility traces to offset the problem of delayed reward. The aim is the design of an agent that is able to learn in an unsupervised manner in a complex environment, eventually taking over tasks that had previously been performed by non-adaptive, deterministic game AI. The preliminary results presented in this paper show the viability of the RL algorithms at learning the selected task. Depending on whether the focus lies on maximizing the reward or on the speed of learning, among the evaluated algorithms one-step Q-learning and Sarsa(λ) prove best at learning to manage combat units.**

I. Introduction

After games become increasingly photorealistic, focus now switches towards computer game AI, just as predicted by Laird and van Lent more than a decade ago [1]. The video game market, already a multi-billon dollar business, becomes more and more pervasive in everyday life. With the current boom in home and portable game consoles and the proliferation of smart phones, the computing power for complex game AI is more accessible than ever. While the focus of commercial game AI development is slowly shifting away from the ever-present A* algorithm, still the best example of academic research that is successful in commercial games, the move towards more complex technologies and away from the deterministic and predictable FSM and decision trees remains very slow.

The lack of enthusiasm for adapting more complex academic AI techniques for commercial games is on the one hand due to the fact that game developers want to avoid unpredictable and thus potentially faulty behavior in their games. On the other hand, a lot of academic research seems far away from everyday game development needs with lots of theoretically plausible components that are hard to realize within the constraints of commercial games. This is where research using commercial games as test bed can help and prove that complex machine learning techniques, such as reinforcement learning, are viable alternatives to the existing mostly deterministic game AI. For AI researchers, commercial games provide rich environments and have started to provide powerful interfaces such as the one used in this paper (Broodwar API (BWAPI)) that enable intricate interaction with the game, basically enabling any contrivable modification.

Stefan Wender and Ian Watson are with The University of Auckland, Department of Computer Science, Auckland, New Zealand; e-mail: swen011@aucklanduni.ac.nz || ian@cs.auckland.ac.nz

Our aim is to evaluate the suitability of reinforcement learning [2], as a machine learning (ML) technique that enables adaptive AI, for creating a learning agent in SC:BW. In order to do this, we use two prominent temporal-difference (TD) learning algorithms both in their simple one-step versions and in their more complex versions that use eligibility traces. The task chosen for this evaluation is managing a combat unit in a small scale combat scenario in SC:BW. Eventually this agent will be the part of a larger hybrid AI solution that addresses the entire SC:BW gameplay problem, managing the different layers of problems existing in a complex commercial RTS game [3]. Our desire is for the technique to be easily scalable for different types of units and even flexible enough to be transferred to different problem domains such as other games or even different sets of problems as described in [1].

II. Related Work

RL in games is a popular area of application for AI research [4], with the technique being well suited to complex game environments. Since Tesauro's landmark use of TD reinforcement learning [5], RL has been used in a variety of different types of AI research both in conventional games and in video games. RL has been used to create adaptive agents in a fighting game [6], to control teams of players in the commercial First-Person Shooter game Unreal Tournament [7], to select city sites in the turn-based strategy game Civilization IV [8] or, together with case-based reasoning (CBR), to control sets of units in the RTS MadRTS in fighting scenarios [9] to name but a few.

Most closely related to this paper is research focused on using RL in unit management scenarios in SC:BW [10], [11]. [10] describes the implementation of a simple, very specialized RL-based agent for micro-managing combat units. The chosen state representation is highly tailored towards the experiments that are run and the size of the state space severely limits the efficiency of the learning agent. An extension of the managed units beyond those used in the experiments does not seem possible without entirely redesigning the RL model. [11] use neural networks (NNs) to approach the problem of state space complexity that is inherent in complex games such as SC:BW. The NN receives the game state as input and then

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

approximates the state-action value function $Q(s_t, a_t)$ to be used for the Sarsa RL algorithm, thus creating neural-fitted Sarsa (NFS). This technique is then shown with sufficient training to significantly outperform the built-in game AI in combat scenarios between a small number (3vs3) of similar units and show promise in a larger scenario of 6vs6. However, the authors also point out that the gains of NFS over normal Sarsa were small. Scalability is also a problem as the learning process was still very slow in the 6vs6 scenario, despite the acceleration by transferring knowledge from the 3vs3 scenario.

III. StarCraft as Environment for Research in Artificial Intelligence

Due to its enormous popularity and the well-documented and well-maintained BWAPI interface, SC:BW has recently seen a large increase in its use as a test bed for AI research. Another reason for SC:BW's popularity as a test bed for research is the fact that it exhibits all the characteristics identified by [3] as interesting AI problem areas in RTS environments. These characteristics include adversarial planning, incomplete information, spatial and temporal reasoning, resource management, pathfinding and being a multi-agent environment that generally allows for machine learning and opponent modeling.

SC:BW has not only been used for research directly focusing on the in-game AI but also for other topics such as to explore its network traffic [12], for user identification through mouse movement patterns [13] and procedural content generation to automatically generate playable SC:BW maps [14].

Due to its large scope and complex problem domain, StarCraft poses an ideal background for research in planning algorithms. In fact, large parts of the AI research using StarCraft as test bed is concerned with different aspects of automated planning: [15] uses Goal-Oriented Action Planning (GOAP) in his planner *StarPlanner*, which makes high-level and mid-level decisions in SC:BW. [16] creates a planning agent capable of playing complete games by splitting the decision-making process into several interconnected parts. [17] builds an autonomous reactive planning agent using the reactive planning language ABL.

Another advantage of using SC:BW is the large amount of game traces in the form of game replays (previously recorded games) that are available online. Replays have been used as the case base for a CBR system that analyzes players' building strategies and uses this knowledge for strategy prediction [18] and to analyze which build order performs best against which other build order [19]. Replays were also analyzed in the context of cognitive research to find correlations between the observable actions of a player and performing successfully in the game [20]. [21] examines a large number of StarCraft replays in order to predict an opponents strategy through their build order. [22] pursues a similar approach of analyzing game logs to create a simple Bayesian model for predicting the buildings a player uses.

IV. Reinforcement Learning

Reinforcement learning is an unsupervised machine learning technique which puts an agent into an unknown environment giving it a set of possible actions and the aim of maximizing a reward. As described in previous sections, RL has been applied successfully to a wide range of problems from varying areas. The characteristics exhibited by RTS games such as adversarial planning, incomplete information combined with spatial and temporal reasoning [3] make RTS games a great test bed for unsupervised machine learning techniques such as RL.

A. Reinforcement Learning Algorithms

The algorithms used throughout this paper are variations of the standard TD reinforcement learning algorithms Q-learning and Sarsa as described in [2] which were adapted for the chosen problem environment. Both the simpler one-step versions of these algorithms as well as variations that use eligibility traces to offset the problem of delayed reward are evaluated.

1) Q-Learning: Q-learning is an off-policy TD algorithm that does not assign a value to states, but to state-action pairs [23]. Since Q-learning works independent of the policy being followed, the learned action value Q function directly approximates the optimal action-value function Q^*. Equation 1 shows the value update function for Q-learning.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma max_a Q(s_{t+1}, a_t) - Q(s_t, a_t) \right]. \quad (1)$$

2) Sarsa: One-step Sarsa is an on-policy TD learning algorithm very similar to Q-learning [24]. The main difference is that it is not necessarily the action with the biggest reward that is used for the next state but the action chosen according to the same policy that led to the present state. Sarsa stands for **S**tate-**A**ction-**R**eward-**S**tate-**A**ction, more specifically the quintuple referred to here is $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Equation 2 shows the value update function for Sarsa.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \quad (2)$$

3) Eligibility Traces: Eligibility traces are a basic mechanism of reinforcement learning that is used to assign temporal credit. This means that it is not only the value for the most recently visited state or state-action pair that is updated but also the value for states or state-action pairs that have been visited within a limited time before. The mechanism assigns a trace parameter to recently visited states. The value of this trace parameter - which decreases each round through the eponymous trace decay parameter λ - determines the amount of reward that the state-action pair receives. For both Sarsa and Q-learning there exist algorithms Sarsa(λ) and Q(λ) that include eligibility traces. There are two main versions of Q(λ) that work differently when it comes to cutting off eligibility traces. Cutting traces becomes necessary as Q-learning is an off-policy algorithm which means that otherwise, once an action has been taken to reach a state which was not part of the initial Q(s,a) calculation, reward would be assigned improperly via eligibility traces.

This is the reason why Watkins's Q(λ) [23] uses

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } (s,a) \neq (s_t, a_t); \\ 0 & \text{if } Q_{t-1}(s_t, a_t) \neq maxQ_{t-1}(s_t, a_t) \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } (s,a) = (s_t, a_t) \\ & \text{and } Q_{t-1}(s_t, a_t) = maxQ_{t-1}(s_t, a_t). \end{cases}$$

to define eligibility traces for state-action pairs. Peng's Q(λ) [25] on the other hand never cuts traces, thus making it basically a hybrid of Watkins's Q(λ) and Sarsa(λ) [2]. Therefore, we decided to omit Peng's Q(λ) from our experimental evaluation and focus on Watkins's Q(λ) and Sarsa(λ).

B. Model

The agent in a reinforcement learning framework makes its decisions based on the state s an environment is in at any one time. If this state signal contains all the information of present and past sensations it is said to have the *Markov property*. If a RL task presents the Markov property it is said to be a *Markov decision process* (*MDP*). A specific MDP is defined by a quadruple $(S, A, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'})$. RL problems are commonly modeled as MDPs, where S is the state space, A is the action space, $\mathcal{P}^a_{ss'}$ are the transition probabilities and $\mathcal{R}^a_{ss'}$ represents the expected reward, given a current state s, an action a and the next state $s_{\prime}$. The MDP is used to maximize a cumulative reward by deriving an optimal policy π according to the given environment.

In order to adapt the RL technique to the given task of micro-managing combat units, a suitable representation has to be chosen. This is especially important since SC:BW is a very complex environment that requires extensive abstraction of the available information in order to enable meaningful learning and prevent an exponential growth of the state- or action space. This would make any meaningful learning in reasonable time impossible.

1) States: States are specific to a given unit at a given time and are designed as a quadruple of values consisting of

- Weapon Cooldown: Is the weapon of this unit currently in cooldown?
- Distance to Closest Enemy: Distance to closest enemy as a percentage of the possible movement distance of this unit within the near future. Distances are grouped into four different groups ranging between $<= 25\%$ and $> 120\%$
- Number of Enemy Units in Range: Number of enemy units within range of this unit's weapon.
- Health: Remaining health of this unit, classified into one four different classes of 25% each (i.e. 0%-25% etc.).

This leads to a possible size of the state space

$$|S| = 32 * (|U_{enemy}| + 1)$$

where U_{enemy} is the set of all enemy units.

2) Actions: There are two possible actions for the units in this model, 'Fight' or 'Retreat'. This set of actions was chosen for several reasons. While there are many more possible actions for an agent playing a complete game of SC:BW, the focus for this task and therefore for this model lay on unit control in small scale combat. This excludes many high level planning actions, such as actions that are given to groups of units. Many units in SC:BW have abilities reaching beyond standard fighting, so-called *special abilities* that can be decisive in battles and which are ignored in this setup. However, the core of any army and therefore the core of any fight will always consist of ranged or melee units that have no special abilities beyond delivering normal damage. These other parts will eventually be managed by other RL-based agents.

Only using 'Fight' and 'Retreat' as actions also leaves out more general actions such as exploring the surroundings or simply remaining idle. Exploration is a crucial part of any SC:BW game but involves many different aspects that are not directly related to combat and will therefore eventually be handled by a completely separate component of the AI agent. The 'Idle' action was originally part of the model but empirical evaluation showed that this only meant that the built-in unit micro management took over, which is actually a mix of both already existing states, 'Fight' and 'Retreat'. Therefore, the learning process was severely impeded by the 'Idle' action receiving a reward signal that was basically a mix of the reward signal of the other two actions.

a) Fight Action: The 'Fight' action is handled by a very simple combat manager. The combat manager determines all enemy units within the agent's unit's range and selects the opponent with the lowest health which can thus be eliminated the fastest. Should no enemy unit be within weapon range while the action is triggered, nothing will happen until the next action has to be chosen.

b) Retreat Action: The 'Retreat' action means for the agent's unit to get away from the source of danger, i.e. enemy units, as fast as possible. A weighted vector is computed that leads away from all enemy units within the range they would be able to travel within the next RL time frame. The influence of one enemy unit on the direction of the vector is determined by the amount of damage it could do to the agent's unit.
To avoid being trapped in a corner, on the edge of a map or against an insurmountable obstacle on the map (SC:BW maps can have up to three different levels of height, separated by cliffs) a repulsive force is assigned to these areas. This force is included into the computation of the vector. If an agent's unit that tries to retreat is surrounded by equally threatening and thus repulsive forces, this means that no or next to no movement will take place.

3) Transition Probabilities: As becomes obvious from the description of S and A, the transition rules are stochastic and, depending on the behavior of the units involved, different subsequent states s_{t+1} can be reached after taking the same action a_t in the same state s_t at different times in the game.

4) Reward: The reward signal is based on the difference in health of both the enemy's and the player's units between two states. The reward is computed as the difference in health of the enemy's units (i.e. the damage done by the agent) minus the difference in the RL agent's unit (i.e. the damage received by the agent).

$$reward_{t+1} = \sum_{i=1}^{m} enemy\_unit\_health_{i_t} - enemy\_unit\_health_{i_{t+1}} - (agent\_unit\_health_t - agent\_unit\_health_{t+1})$$

This means the agent will measure its success in the amount of damage it is able to deal while trying to retain as much of its own health as possible.

V. Empirical Evaluation and Results

For the evaluation of the performance of the selected RL algorithms we designed a small scale combat scenario in SC:BW that allows the RL agents to show their ability to learn in an unsupervised environment. The scenario consists of one combat unit controlled by the RL agent fighting a group of enemy units spread out around the starting location of the RL agent as can be seen in figure 1. The RL agent unit has the advantage of superior speed, superior range and - by a small margin - superior firepower in comparison to a single enemy unit. However, when fighting more than one enemy unit, the agent's single unit easily loses. Preliminary tests showed that using only the built-in game AI, the single unit quickly loses in this scenario every time. The aim for the RL agent is to learn to exploit its advantages in terms of speed and range by adopting so-called 'kiting', a form of hit and run: getting and staying out of range of enemy units while firing at them from a safe distance. An episode in our experiment concludes when either the agent's unit or all the enemy's units have been eliminated.

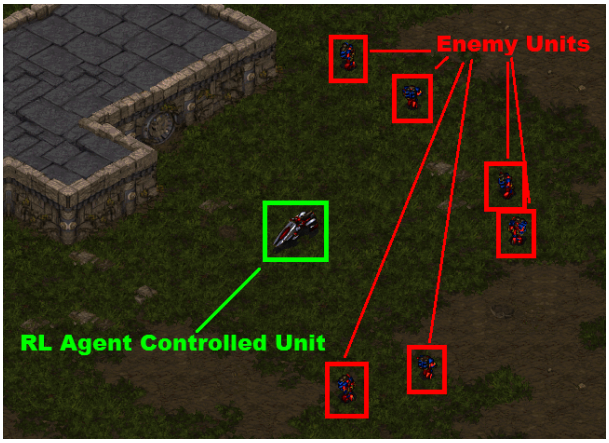

Fig. 1. Initial unit positioning for the experimental evaluation

A. Algorithm

SC:BW, like most other games falling into the RTS game category, emulates 'real-time' gameplay. This means, that while internally the game is sill computed on a turn-by turn basis, these turns happen so quickly that human players perceive the game as being continuous. On the one hand, this limits the computing power that can be used by any algorithms since the player still has to perceive the environment as being real-time. This is especially important for an online learning technique such as RL which runs updates every time frame. On the other hand, RL techniques are usually designed to work with concrete time steps. One possible translation of the RL model into SC:BW would have been to use a fixed number of SC:BW frames as one RL time step. However, this proved problematic as the 'Fight' action for instance varies in duration between units and also depending on the physical positions of units on the map. Therefore, we chose to define one RL time step simply as the time it takes for that action to be finished. For the 'Retreat' action this meant that units would retreat into the chosen direction at maximum speed for a fixed amount of time. A high-level view of the steps involved in integrating our agent into SC:BW can be seen in figure 2.

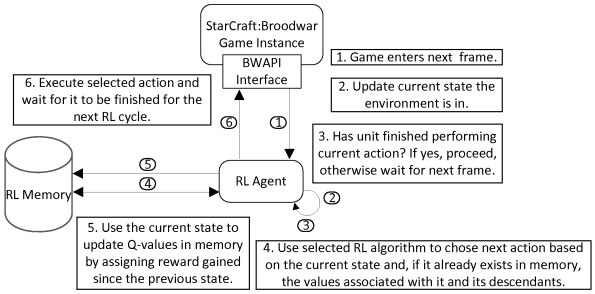

Fig. 2. StarCraft:BroodWar RL algorithm integration

B. Experimental Setup

We ran each of the RL algorithms for 1,000 episodes during which the memory of the agent was not reset, i.e. after 1,000 games the memory was wiped and the agent started learning from scratch. We repeated these 1,000 episodes 100 times for each algorithm in order to gain conclusive insights into their performance. For all algorithms the agent followed an ϵ-greedy policy. However, the value of ϵ was set to decline from its initial value of 0.9 to reach zero at the end and thus make the action selection process more and more greedy towards the end. This would enable the RL agent to always follow Q^* at the end.

After the 1,000-episode runs we repeated the process with shorter 500-episode runs, again with diminishing ϵ-greedy policy. This was done in order to gain a better understanding of algorithm performance in shorter terms and their speed of convergence towards an optimal policy. Altogether, each of the RL algorithms played 100,000 games for the 1,000-episode runs and 50,000 games for the 500-episode runs. The values chosen for the experimental setup are $\alpha = 0.05$, $\gamma = 0.9$ and, for those algorithms using eligibility traces, $\lambda = 0.9$. These values result in a slow learning process which hugely discounts possible future rewards. For algorithms using eligibility traces, the slow decay of these traces results in actual future rewards playing a big role for previously visited states.

C. Results

Figures 3 to 8 show the results of the experimental evaluation. The diagrams show averaged values where each data point represents the average of ten values from the 500-game or 1,000-game runs, leading to 50 or 100 data points respectively.

We measured the development of the overall reward gained by the RL agent during one game (figures 3 and 5). The overall reward is the total sum of all rewards the RL agent achieves during one game. An optimal performance, i.e. defeating all enemy units without sustaining any damage to the agents own unit, would result in a total reward of 300. Since the eventual aim for the agent will be to win an entire game, we also recorded the percentage of games the agent is able to win, i.e. eliminate all opponents without dying (figures 4 and 6). This value is closely tied to the overall reward, but not entirely similar, something that can be observed when comparing figures 5 and 6. Furthermore, we computed the standard deviation as a measurement of the variance in results (figures 7 and 8).

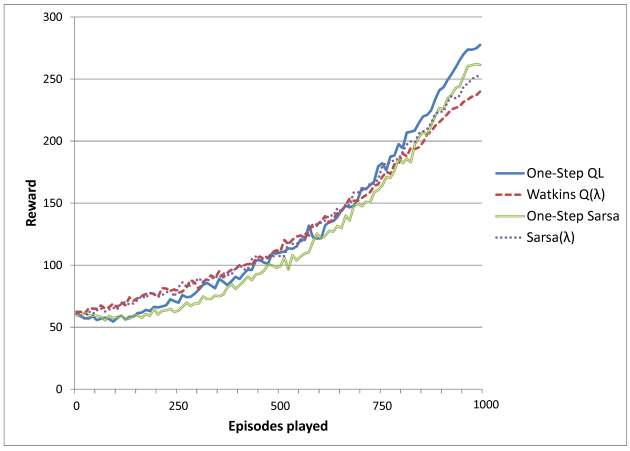

Fig. 3. Development of Total Reward for 1000 Episodes Played

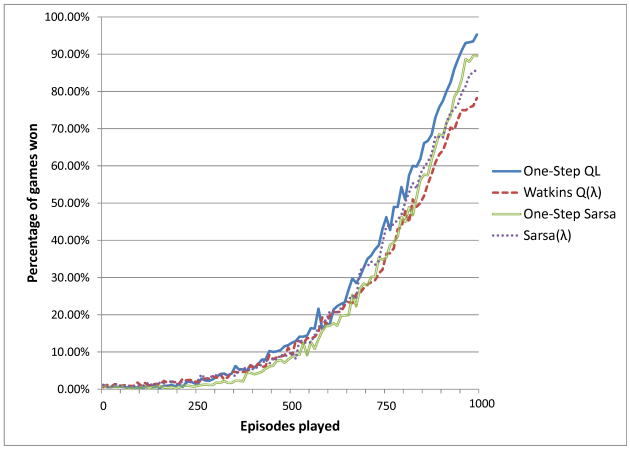

Fig. 4. Percentage of Games Won for 1000 Episodes Played

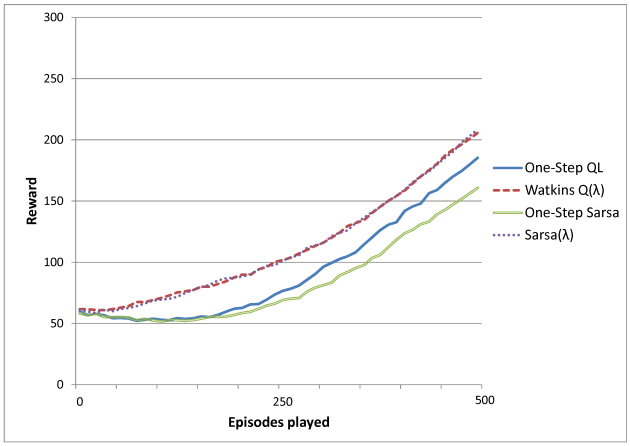

Fig. 5. Development of Total Reward for 500 Episodes Played

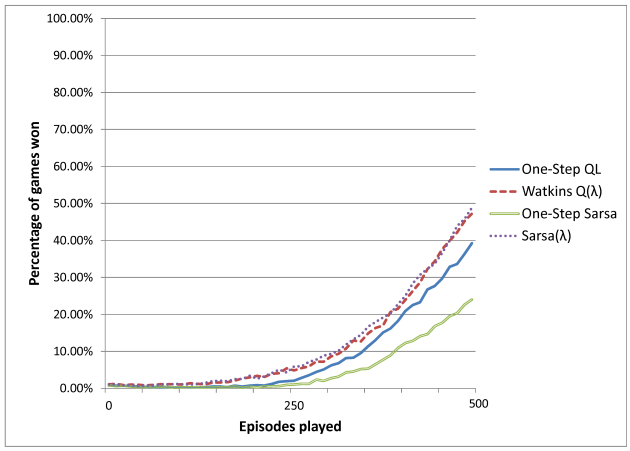

Fig. 6. Percentage of Games Won for 500 Episodes Played

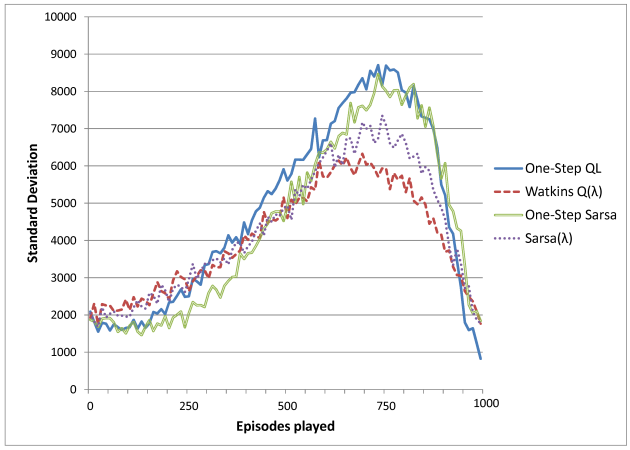

Fig. 7. Standard Deviation for 1000 Episodes Played

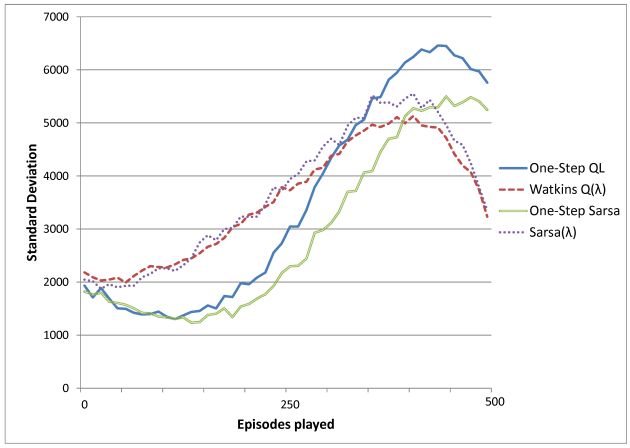

Fig. 8. Standard Deviation for 500 Episodes Played

VI. Discussion and Future Work

After 1,000 episodes of learning, all four tested algorithms manage to achieve easy wins when following their most greedy learned policy. The total reward is close to maximum value of 300 for all algorithms towards the end. This means that all algorithms manage to learn a policy close to the optimal policy π in 1,000 episodes. Furthermore, even the speed of learning seems to differ little between the different algorithms. The reward for the one-step versions of the algorithms slightly declines at first, however starts to grow equally quick to that of Watkins's Q(λ) and Sarsa(λ) after about one third of the experimental run. Eventually both one-step Q-learning and one-step Sarsa manage to gain more reward than their counterparts that are based on eligibility traces. Using the best possible learned policy at the very end, one-step Q-learning achieves an average reward of $\tilde{2}75$, slightly outperforming one-step Sarsa with $\tilde{2}62$, Sarsa(λ) with $\tilde{2}52$ and, a little behind, Watkins's Q(λ) with $\tilde{2}37$. The results also show that this is the biggest difference between the best and the worst performing algorithm at any point throughout the whole 1,000 episodes.

The behavior of the standard deviations for 1,000 episodes, depicted in figure 7, differs more between algorithms than the results for total reward and games won. Ideally we would expect a large standard deviation at the start, when a good policy is still unknown. This initially large standard deviation would then diminish towards the end when the agent uses more and more greedy selections which finally lead to only following their approximation of the optimal policy π.
The results show the same general behavior for all algorithms: The initial standard deviation rises to a peak in the later stages of the 1,000-episode run and finally sharply drops below even the initial value when approaching a purely greedy policy. The reason for this behavior is the greater variance of possible results once the agent has acquired a certain amount of knowledge. It therefore sometimes selects very good policies while on the other hand still following a partially explorative policy that can lead to low reward. The initial standard deviation results from the near-random policy that the agent follows at the start of an experimental run. Towards the end the agent quickly approaches the point where it always follows the same policy - the remaining variance is a result of the non-deterministic state transitions resulting from the complexity of the game.

The standard deviation for the one-step versions of the algorithms marginally declines at the beginning of the evaluation run. After about one third of the run the standard deviation for one-step Q-learning and one-step Sarsa starts to grow rapidly, quickly surpassing that of Watkins's Q(λ) and Sarsa(λ), which has been growing from the beginning but at a less rapid pace. This behavior bears a similarity to that of the reward signal, thus indicating the connection between acquired knowledge and a high standard deviation that was mentioned in the previous paragraph.
Among the algorithms, one-step Q-learning shows the smallest standard deviation towards the end while having the largest deviation of all algorithms (though not by far) at its peak. This means that one-step Q-learning gets closest to repeatedly following an optimal policy at the end - something which is in line with what figure 3 shows. The standard deviation for one-step Sarsa shows a similar behavior to that of one-step Q-learning with a slightly less pronounced peak. The lowest peak for the standard deviation is shown by Watkins's Q(λ), otherwise it behaves similar to the other algorithms. The randomness in exploration is the same for all algorithms, therefore the difference must lie in the best learned policy. As the development of its total reward already showed that Watkins's Q(λ) performed slightly worse than the other algorithms, the behavior of the standard deviation now gives a hint where the problem lies. The behavior of the standard deviation is practically similar for all algorithms in the first half of the experimental run and the standard deviation of Watkins's Q(λ) only starts to grow less fast in comparison to the other algorithms after about 500 runs. Therefore, this must be the point where Watkins's Q(λ) starts to learn less than the other algorithms. Further experiments will have to show what exactly makes the difference here.

The performance in terms of reward gained and winning games over 1,000 episodes is very similar between all algorithms, only showing minimal differences in their accuracy. Therefore, the results of the 500-episode runs are important to compare the learning capabilities of the different algorithms in terms of speed.
Most notable is that, while a learning process definitely took place, none of the algorithms managed to achieve the optimal results as seen in the 1,000-episode runs. Both of the algorithms that use eligibility traces, Watkins's Q(λ) and Sarsa(λ), show a similar, comparably strong performance. In the end of the 500-episode runs, following their best learned policy, they achieve an average reward of about 200, about two thirds of the maximum possible reward. Compared to this, one-step Q-learning achieves about 10% less reward overall and one-step Sarsa about 25% less than both best performing algorithms. Furthermore, there is a distinct difference in the learning curves between the algorithms that use eligibility traces and their one-step pendants. Both one-step Q-learning and one-step Sarsa show a slightly declining curve at first, very similar to that exhibited by all algorithms in the longer 1,000-episode runs. Subsequently, that curve rises again, growing nearly linear with one-step Q-learning changing quicker than one-step Sarsa from decline to growth. This growth eventually results in the difference in overall reward between one-step Q-learning and one-step Sarsa at the end.
The diagram for the algorithms using eligibility traces indicate a faster learning process compared to the one-step algorithms. While it takes one-step Q-learning and one-step Sarsa more than 100 episodes to start increasing the obtained reward, Watkins's Q(λ) and Sarsa(λ) nearly instantly (there is a minimal decrease in reward at first) start increasing their total reward. This is the same behavior that is apparent in the longer runs of 1,000 episodes, only more distinct due to larger differences in values and lower overall rewards. The instant growth of reward for Watkins's Q(λ) and Sarsa(λ) basically shows the effectiveness of eligibility traces in terms of speeding up the learning process.
Figure 8 shows how the behavior of the standard deviation is quite different between the algorithms. The standard deviation for all four algorithms is initially identical to that in figure 7: the standard deviation for one-step Q-learning and one-step Sarsa shows a decline at first before starting to rapidly grow and eventually surpassing that of Watkins's Q(λ) and Sarsa(λ). However, compared to that of 1,000-episode runs it is different towards the end when the policy gets more and more greedy for all algorithms. Watkins's Q(λ) and Sarsa(λ) show a drop similar - though less distinct - to that of 1,000-episode runs. The one-step versions of the algorithms however, show much less of a drop and start to decline much later. The standard deviation for Sarsa(λ) even seems to stay linear before declining a little in the last few episodes. These results indicate the lack of a sufficiently advanced policy because the agent so far has been unable to learn which are the best states/actions and therefore is stuck with a sub-par policy. The development of the overall reward for these two algorithms confirms this conclusion.

The empirical evaluation shows that all RL algorithms are suitable to learn the task we chose for our evaluation. One-step Q-learning demonstrates the strongest performance in terms of accuracy, given enough episodes to learn the required task. One-step Sarsa is a close second. Watkins's Q(λ) and Sarsa(λ) also display good results in terms of accuracy, however their strength seems to lie in learning fast and, ensuing from this, speed of convergence

towards an optimal policy π.

In future work we are planning to extend, refine and reuse the findings of this paper in several ways. While eligibility traces help to speed up the convergence towards an optimal policy as expected, this happened at a tradeoff of diminishing overall reward. Ideally this tradeoff would not be necessary, therefore we will investigate if possible changes to the algorithm or to the model will enable Watkins's Q(λ) and Sarsa(λ) to perform at the same level as or even higher than their one-step counterparts.

The values chosen for α, γ and λ in the experimental setup are based on common values for RL experiments [2] but might not be optimal for our specific problem. We are therefore planning to use a genetic algorithm to optimize these settings.

Furthermore, the solution developed in this paper is intended to be the first part of a larger RL-based agent that is able to play entire games of SC:BW. Due to the game's large complexity this will not be possible by one agent only, but will have to be handled by different agents handling different layers of the game [16] [26]. The next step will be to introduce an agent that controls the next higher level of complexity and coordinates a team of units that are themselves controlled by the agents introduced in this paper. Simple RL will not be enough to handle all the different tasks in SC:BW, some of which are very complex [3]. Therefore, we are planning to also use other ML techniques such as CBR to create hybrid approaches [27] for the different problems inherent in SC:BW.

VII. Conclusions

The empirical evaluation over 1,000 episodes shows that an AI agent using RL is able to learn a strategy that beats the built-in game AI in the chosen small scale combat scenario in approximately 100%. This impressive win rate is achieved by all RL algorithms that we evaluated and hints at the extensive possibilities of our RL agent.

As our aim was to evaluate the usability of RL in SC:BW as a representative commercial RTS, we also have to take into account the efficiency of the algorithms and thus the time it takes the agent to develop a usable strategy. This is very important as, should these types of ML algorithms ever be considered for application in commercial games, the learning process has to be on the one hand quick while on the other hand avoiding major drops in performance. Our experiment over half the number of episodes showed that a quicker learning curve is indeed possible but currently comes at the price of a decrease in obtained reward. In the case of the best-performing one-step Q-learning this shortening of the learning phase even leads to a performance drop of nearly 40% .

Therefore, further research is clearly needed before it will be feasible to create a powerful standard game AI in a commercial game such as SC:BW using RL. However, our results are very promising in terms of overall performance. Furthermore, those results also show many possible ways of easily reusing our current findings to advance the development of a RL agent that is able to play entire games of SC:BW.

References

[1] J. Laird and M. van Lent, "Human-level ai's killer application: Interactive computer games," *AI Magazine*, vol. Summer 2001, pp. 1171–1178, 2001.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[3] M. Buro, "Real-time strategy gaines: a new ai research challenge," in *Proceedings of the 18th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 2003, pp. 1534–1535.

[4] I. Szita, "Reinforcement learning in games," *Reinforcement Learning*, pp. 539–577, 2012.

[5] G. Tesauro, "Temporal difference learning of backgammon strategy," in *Proceedings of the 9th International Conference on Machine Learning 8*, 1992, pp. 451–457.

[6] G. Andrade, G. Ramalho, H. Santana, and V. Corruble, "Automatic Computer Game Balancing: A Reinforcement Learning Approach," in *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2005, pp. 1111–1112.

[7] M. Smith, S. Lee-Urban, and H. Muñoz-Avila, "Retaliate: Learning winning policies in first-person shooter games," in *Proceedings of the Seventeenth Innovative Applications of Artifcial Intelligence Conference (IAAI-07)*. AAAI Press, 2007, pp. 1801–1806.

[8] S. Wender and I. Watson, "Using Reinforcement Learning for City Site Selection in the Turn-Based Strategy Game Civilization IV," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games (CIG'08) (to appear)*, 2008, pp. 372–377.

[9] M. Molineaux, D. Aha, and P. Moore, "Learning continuous action models in a real-time strategy environment," in *Proceedings of the Twenty-First Annual Conference of the Florida Artificial Intelligence Research Society*, 2008, pp. 257–262.

[10] A. Micić, D. Arnarsson, and V. Jónsson, "Developing game ai for the real-time strategy game starcraft," Reykjavik University, Tech. Rep., 2011.

[11] A. Shantia, E. Begue, and M. Wiering, "Connectionist reinforcement learning for intelligent unit micro management in starcraft," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2011*, 2011.

[12] A. Dainotti, A. Pescape, and G. Ventre, "A packet-level traffic model of starcraft," in *Hot Topics in Peer-to-Peer Systems, 2005. HOT-P2P 2005. Second International Workshop on*. IEEE, 2005, pp. 33–42.

[13] R. Kaminsky, M. Enev, and E. Andersen, "Identifying game players with mouse biometrics," University of Washington, Tech. Rep., 2008.

[14] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, and G. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 265–272.

[15] P. Peikidis, "Demonstrating the use of planning in a video game," Master's thesis, University of Sheffield, 2010.

[16] F. Safadi and D. Ernst, "Organization in ai design for real-time strategy games," Master's thesis, Universit de Lige, 2010.

[17] B. Weber, M. Mateas, and A. Jhala, "Applying goal-driven autonomy to starcraft," in *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[18] J. Hsieh and C. Sun, "Building a player strategy model by analyzing replays of real-time strategy games," in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. IEEE, 2008, pp. 3106–3111.

[19] J. Kim, K. Yoon, T. Yoon, and J. Lee, "Cooperative learning by replay files in real-time strategy game," *Cooperative Design, Visualization, and Engineering*, vol. 6240/2010, pp. 47–51, 2010.

[20] J. Lewis, P. Trinh, and D. Kirsh, "A corpus analysis of strategy video game play in starcraft: Brood war," in *The Annual Meeting Of The Cognitive Science Society (COGSCI 2011)*, 2011.

[21] B. Weber and M. Mateas, "A data mining approach to strategy prediction," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 140–147.

[22] G. Synnaeve and P. Bessiere, "A bayesian model for plan recognition in rts games applied to starcraft," in *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2011)*, 2011.

[23] C. Watkins, "Learning from Delayed Rewards," Ph.D. dissertation, University of Cambridge, England, 1989.

[24] G. A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Cambridge University Engineering Department, Tech. Rep. CUED/F-INFENG/TR 166, 1994. [Online]. Available: citeseer.ist.psu.edu/rummery94line.html

[25] J. Peng and R. J. Williams, "Incremental Multi-Step Q-Learning," in *Machine Learning*. Morgan Kaufmann, 1994, pp. 226–232.

[26] B. Weber, P. Mawhorter, M. Mateas, and A. Jhala, "Reactive planning idioms for multi-scale game ai," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 115–122.

[27] B. Weber and S. Ontanón, "Using automated replay annotation for case-based planning in games," in *ICCBR Workshop on CBR for Computer Games (ICCBR-Games)*, 2010.