Using Plan-Based Reward Shaping To Learn Strategies in StarCraft: Broodwar

Kyriakos Efthymiadis
Department of Computer Science
University of York
ke517@york.ac.uk

Daniel Kudenko
Department of Computer Science
University of York
daniel.kudenko@york.ac.uk

***Abstract*—StarCraft: Broodwar (SC:BW) is a very popular commercial real strategy game (RTS) which has been extensively used in AI research. Despite being a popular test-bed reinforcement learning (RL) has not been evaluated extensively. A successful attempt was made to show the use of RL in a small-scale combat scenario involving an overpowered agent battling against multiple enemy units [1]. However, the chosen scenario was very small and not representative of the complexity of the game in its entirety. In order to build an RL agent that can manage the complexity of the full game, more efficient approaches must be used to tackle the state-space explosion. In this paper, we demonstrate how plan-based reward shaping can help an agent scale up to larger, more complex scenarios and significantly speed up the learning process as well as how high level planning can be combined with learning focusing on learning the Starcraft strategy, *Battlecruiser Rush*. We empirically show that the agent with plan-based reward shaping is significantly better both in terms of the learnt policy, as well as convergence speed when compared to baseline approaches which fail at reaching a good enough policy within a practical amount of time.**

I. Introduction

The video game market, aided by hardware advancements, has been very rapidly expanding over the past 10 years and has now become a part of everyday life. With games becoming increasingly more realistic in terms of graphics, it has become apparent that it is not only graphics that make a game feel real, but also the built in AI. Commercial games have been using the A^* algorithm when shipping titles mainly due to its stable behaviour and minimal use of resources. However, given the increased power of game consoles, personal computers and even smart-phones, the focus of game AI is very slowly shifting toward more adaptive approaches.

Reinforcement learning (RL) is one such adaptive approach which has proven to be very successful when an agent needs to act and improve in a given environment. The agent receives feedback about its behaviour in terms of rewards through constant interaction with the environment. Traditional reinforcement learning assumes the agent has no prior knowledge about the environment it is acting on. As a result, when scaling up to more complex scenarios the agent is greatly affected by the state-space explosion and fails to learn the given task within a practical time-frame.

In many cases domain knowledge of the RL tasks is available, and can be used to improve the learning performance. In earlier work on *knowledge-based reinforcement learning* (KBRL) [2], [3] it was demonstrated that the incorporation of domain knowledge in RL via reward shaping can significantly improve the speed of converging to an optimal policy. Reward shaping is the process of providing prior knowledge to an agent through additional rewards. These rewards help direct an agent's exploration, minimising the number of sub-optimal steps it takes and so directing it towards the optimal policy quicker. Plan-based reward shaping [2] is a particular instance of knowledge-based RL where the agent is provided with a high level STRIPS [4] plan which is used in order to guide the agent to the desired behaviour.

In this paper, we demonstrate the benefits of using plan-based reward shaping to create an RL agent within the game of SC:BW. The agent's task is finding the correct build order in order to be able to construct a specific unit. The agent using plan-based reward shaping is then compared to a baseline RL agent with no domain knowledge. We demonstrate empirically that the shaped agent is performing significantly better in terms of discounted goal reward, and is able to find the optimal policy within a practical amount of time while the agent without shaping is daunted by the complexity of the task and fails to learn a good enough policy. The results highlight the importance of using KBRL in order to create an adaptive solution that would be able to manage the complexity of the entire game of SC:BW. In addition, it is shown how high level planning can be used to guide low level behaviour.

II. StarCraft

StarCraft is a very popular commercial RTS game which has been used in AI research extensively with numerous competitions running in parallel to major game conferences. This was made possible with the development of the BWAPI framework which allows integration with the SC:BW game engine.

As with every RTS game, the gameplay revolves around resource management, and building construction. Resource management is very important in every RTS game since it is what allows the players to build advanced units. In StarCraft the two major resources are *vespene gas* and *minerals*. Worker units are tasked with collecting those resources.

Once enough resources have been gathered, the player can decide how to expand. Different build orders allow the production of different units and result in different strategies in order to beat the opponent which can either be another player, or the built in game AI. Many different strategies have been developed throughout the years by players, some focusing

978-1-4673-5311-3/13/$31.00 ©2013 IEEE

on exploration and large armies, while others on speed and efficiency.

III. Background

A. Reinforcement Learning

Reinforcement learning is a method where an agent learns by receiving rewards or punishments through continuous interaction with the environment [5]. The agent receives a numeric feedback relative to its actions and in time learns how to optimise its action choices. Typically reinforcement learning uses a Markov Decision Process (MDP) as a mathematical model [6].

An MDP is a tuple $\langle S, A, T, R \rangle$, where S is the state space, A is the action space, $T(s, a, s') = Pr(s'|s, a)$ is the probability that action a in state s will lead to state s', and $R(s, a, s')$ is the immediate reward r received when action a taken in state s results in a transition to state s'. The problem of solving an MDP is to find a policy (i.e., mapping from states to actions) which maximises the accumulated reward. When the environment dynamics (transition probabilities and reward function) are available, this task can be solved using dynamic programming [7].

When the environment dynamics are not available, as with most real problem domains, dynamic programming cannot be used. However, the concept of an iterative approach remains the backbone of the majority of reinforcement learning algorithms. These algorithms apply so called temporal-difference updates to propagate information about values of states, $V(s)$, or state-action pairs, $Q(s, a)$. These updates are based on the difference of the two temporally different estimates of a particular state or state-action value. The SARSA algorithm is such a method [5]. After each real transition, $(s, a) \rightarrow (s', r)$, in the environment, it updates state-action values by the formula:

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

where α is the rate of learning and γ is the discount factor. It modifies the value of taking action a in state s, when after executing this action the environment returned reward r, moved to a new state s', and action a' was chosen in state s'.

It is important whilst learning in an environment to balance exploration of new state-action pairs with exploitation of those which are already known to receive high rewards. A common method of doing so is $\epsilon-$greedy exploration. When using this method the agent explores, with probability ϵ, by choosing a random action or exploits its current knowledge, with probability $1 - \epsilon$, by choosing the highest value action for the current state [5].

Typically, reinforcement learning agents are deployed with no prior knowledge. The assumption is that the developer has no knowledge of how the agent(s) should behave. However, more often than not, this is not the case. Attention is shifting towards what we call *knowledge-based reinforcement learning*, an area where this assumption is removed and informed agents can benefit from prior knowledge.

B. Reward Shaping

One common method of imparting knowledge to a reinforcement learning agent is reward shaping. In this approach, an additional reward representative of prior knowledge is given to the agent to reduce the number of suboptimal actions made and so reduce the time needed to learn [8], [9]. This concept can be represented by the following formula for the SARSA algorithm:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + F(s, s') + \gamma Q(s', a') - Q(s, a)] \quad (2)$$

where $F(s, s')$ is the general form of any state-based shaping reward.

Even though reward shaping has been powerful in many experiments it quickly became apparent that, when used improperly, it can change the optimal policy [9]. To deal with such problems, potential-based reward shaping was proposed [8] as the difference of some potential function Φ defined over a source s and a destination state s':

$$F(s, s') = \gamma\Phi(s') - \Phi(s) \quad (3)$$

where γ must be the same discount factor as used in the agent's update rule (see Equation 1).

Potential-based reward shaping, defined according to Equation 3, has been proven to not alter the optimal policy of a single agent in both infinite- and finite- state MDPs [8] .

More recent work on potential-based reward shaping, has removed the assumptions of a static potential function from the original proof with the existing guarantees maintained even with a dynamic potential function [10].

C. Plan-Based Reward Shaping

Reward shaping is typically implemented bespoke for each new environment using domain-specific heuristic knowledge [3], [9] but some attempts have been made to automate [11], [12] and semi-automate [2] the encoding of knowledge into a reward signal. Automating the process requires no previous knowledge and can be applied generally to any problem domain. Semi-automated methods require prior knowledge to be put in but then automate the transformation of this knowledge into a potential function.

Plan-based reward shaping, an established semi-automated method, generates a potential function from prior knowledge represented as a high level STRIPS plan. STRIPS is a highly popular and widely used and studied formalism used to express automated planning instances. We briefly explain the STRIPS formalism and refer the reader to [4] for further details, as well as a large collection of books, papers etc. on this topic.

The description of the planning problem in STRIPS includes the operators, actions, which specify the behaviour of the system, and start and goal states [4]. Actions have a set of preconditions which have to be satisfied for the action to be executable, and a set of effects which are made true or false by executing the action. The start state contains a set of conditions which are initially true, and the goal state consists of conditions which have to be true or false for the state to be classified as a goal state.

The output of the planner is the sequence of actions, that will satisfy the conditions set at the goal state. The STRIPS plan is translated from an action-based, to a state-based representation so that, whilst acting, an agent's current state can be mapped to a step in the plan[1], as illustrated in Figure 1.

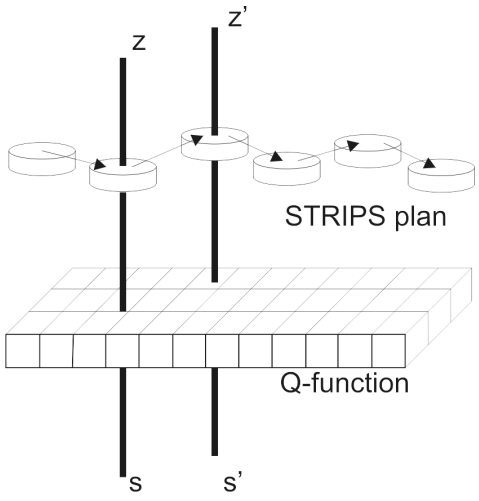

Fig. 1. Plan-Based Reward Shaping.

The potential of the agent's current state then becomes:

$$\Phi(s) = CurrentStepInPlan * \omega \quad (4)$$

where $CurrentStepInPlan$ is the corresponding state in the state-based representation of the agent's plan and ω is a scaling factor.

To preserve the theoretical guarantees of potential-based reward shaping, the potential of all terminal states is set to zero.

These potentials are then used as in Equation 3 to calculate the additional reward given to the agent and so encourage it to follow the plan without altering the agent's original goal. The process of learning the low-level actions necessary to execute a high-level plan is significantly easier than learning the low-level actions to maximise reward in an unknown environment and so with this knowledge agents tend to learn the optimal policy quicker. Furthermore, as many developers are already familiar with STRIPS planners, the process of implementing potential-based reward shaping is now more accessible and less domain specific [2].

IV. StarCraft Scenario

RL has been succesfully used in many games like Backgammon [13], Tetris [14], Unreal Tournament [15] and more. However its performance has not been studied extensively in StarCraft. Of particular interest in the context of this paper is the application of RL in a small-scale combat scenario in the SC:BW [1]. In [1] the scenario involved an overpowered unit (the RL agent), fighting against a group of enemy units spread around the agent. The agent benefited from superior fire-power, speed and range when compared to a single enemy unit. The results showed that the agent quickly learnt how to utilize the "hit and run" strategy, and managed to destroy all enemy units in most of the experiments. These results show the use of RL in SC:BW and the authors aim at creating a hybrid agent that would manage the complexity of the entire game.

Despite the promising results, the state-space of the chosen environment is very small and is not representative of the complexity of the game in its entirety. When trying to scale to larger problem domains, a more efficient design is needed, in order to tackle the state-space explosion, than that of straightforward RL. One method that can achieve this is plan-based reward shaping [2].

In order to demonstrate the use of plan-based reward shaping, we have chosen to evaluate our agent in a more realistic, more complex scenario. The chosen scenario involves the creation of an RL build-order manager that aims at learning part of a player strategy often used in SC:BW, the *Terran Battlecruiser Rush* strategy. The source of the high level strategy can be expert knowledge provided by players, or could come from a high level planning system. The Battlecruiser rush strategy dictates the clever use of resources in order to build the *Battlecruiser* unit as fast as possible while using worker units as support. The goal of this strategy is to march toward the enemy as soon as the Battlecruiser has been constructed so as to surprise the opponent who will most likely be poorly defended. Although this scenario is focused at the *Terran* race, the method can be adapted to any of the races in the game. A model of the agent's learning goal, state, and action space is presented in Section V.

There have been other approaches on using RL in RTS games such as $CLASS_{Q-L}$ [16], in which learning occurs for each class of units in the game in order to speed up the learning process, transfer learning [17], which uses a hybrid case-based reasoning/RL approach in order to learn and reuse policies across multiple environments, and more. In this paper we present how the learning process can be sped up by the use of reward shaping and we believe that our method could be used on top of other RL methods like [16] and [17] to learn policies faster in large state-spaces.

V. Design

Start state: The agent is modelled at a starting position controlling 1 SCV (Terran worker unit), 1 Refinery (used by the Terran race to collect vespene gas), 1 Command Center (Terran base), 1 Resource Depot (used to increase the amount of minerals and vespene gas the agent can collect as well as the amount of units it is allowed to create) and 6 Mineral Fields spread around the starting area. Figure 2 shows the agent's starting state configuration. This configuration creates eighteen slots which can be filled by either worker or combat units i.e. up to 18 units can be trained.

Learning objective: The task that the agent has to learn is that of a strategy often employed by players in SC:BW, the *Battlecruiser rush* strategy. More specifically, this strategy suggests creating one Battlecruiser unit as fast as possible along with several worker units that will act as support for the Battlecruiser, mostly by just being set to repair the damage caused by the opponent. The agent has to learn the

[1]Please note that one step in the plan will map to many low level states. Therefore, the agent must learn how to execute this plan at the low level.

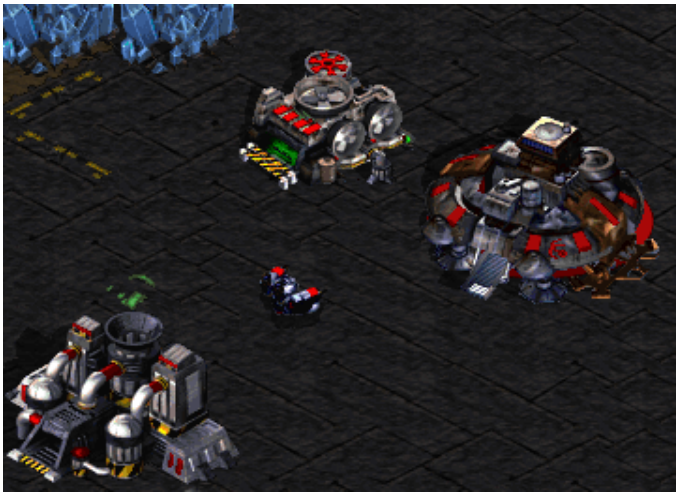

Fig. 2. Agent starting configuration.

optimal building order construction to achieve the creation of the Battlecruiser, while at the same time learn the optimal number of worker units needed in order to have a high enough rate of incoming supplies, to minimise the time to build the Battlecruiser unit. The correct build order, taking into account the constraints of required buildings, is shown in Listing 2 and is *Barracks→Factory→Science Facility→Physics Lab→Starport→Control Tower*. Having constructed these building allows the creation of the Battlecruiser unit.

State-space: Designing the state-space is one of the most important parts in the development of any RL agent. While one can use the raw information contained in the SC:BW environment, it not practical as the amount of information that the agent will have to deal with is so massive, that it would eventually inhibit learning altogether. Therefore the raw information needs to be transformed into a useful abstraction relevant to the task the agent has to tackle. We have chosen the following abstraction:

- *vespene Gas:* the amount of vespene gas the agent has in its possession partitioned in buckets of 50 i.e. 0-50, 50-100. . . >400. We have chosen this design because most units and buildings in SC:BW have a cost which increases by 50 or 100 depending on their level. We have chosen to treat the region > 400 as constant since there are very few units that have a cost of 400 supplies to build. Table I shows the cost of the units that the agent will need to construct and the slots that each unit fills.
- *Minerals:* the amount of minerals the agent has in its possession partitioned in the same manner as vespene gas which was presented previously.
- *Units Built:* the types of units, including buildings, that the agent has under control. These types include the *Factory, Barracks, Starport, Control Tower, Science Facility, Physics Lab* and *Battlecruiser* units. These types are imperative to the creation of the Battlecruiser unit which is the agent's learning task. As a result it is very important information that needs to be included to state-space abstraction.
- *Workers*: the number of worker units the agent has under control. Including this information allows the agent to reason about the optimal number of worker units needed in order to have a supply flow high enough, to allow for the fastest creation of the Battlecruiser.

TABLE I. UNIT COST RELEVANT TO AGENT'S LEARNING OBJECTIVE.

Unit Type	*Minerals*	*vespene Gas*	*Slots Taken*
Factory	200	100	-
Barracks	150	-	-
Starport	150	100	-
Control Tower	50	50	-
Science Facility	100	150	-
Physics Lab	50	50	-
SCV	50	-	1
Battlecruiser	400	300	6

Action-space: The agent can perform a total of ten actions which can be divided into two categories, resource collection and unit construction. Before moving on to the description of those actions, it is worth noting that any chosen action may take a varying amount of time to complete in the game e.g. collecting supplies might take less time than building a specific unit. As a result we define a RL time-step to be the time it takes for an action to be completed.

- *Collect:* the agent can choose to collect either *minerals*, or *vespene gas*. Choosing any of those actions will result in all the worker units the agent controls to move to the closest resource and start collecting supplies. Given that actions to collect gas or minerals are designed to not have a clear end in the game engine i.e. a collect action will continue until a new action is chosen, we set the action as completed after 500 frames. This amounts in a total of two actions for resource collection.
- *Construct:* the agent can choose to build a specific unit. This can be either a building, or a combat/worker unit. The possible units that the agent can choose to construct are *Factory, Barracks, Starport, Control Tower, Science Facility, Physics Lab, SCV worker* and *Battlecruiser*. This amounts in a total of eight actions for construction.

While there are many more actions that an agent can perform when playing the entire game, they are irrelevant to the task at hand and therefore do not need to be included as an option for the agent. Moreover, a different approach could be designed to include parallelism of actions by using their cross-product, however it is a trade off between action-space size and action efficiency.

Reward function: The agent is given a numeric reward of 1000 every time it manages to create the Battlecruiser unit and 0 otherwise. Once the Battlecruiser unit has been built, the episode ends and the agent returns at the starting state.

VI. EVALUATION OF BASELINE AGENT WITHOUT SHAPING

In order to assess the performance of the agent without shaping, a series of experiments were conducted using the

game model that was presented in Section V.

The agent implemented SARSA with $\epsilon-$greedy action selection. For all experiments, the agent's parameters were set such that $\alpha = 0.1$, $\gamma = 0.99$, and $\epsilon = 0.3$ linearly decreasing during an experiment. Decreasing ϵ results in the agent being very explorative at the start of the experiment, and slowly shifting to a more greedy action selection near the end. Each experiment lasted for 500 episodes i.e. games, and was repeated a total of 30 times.

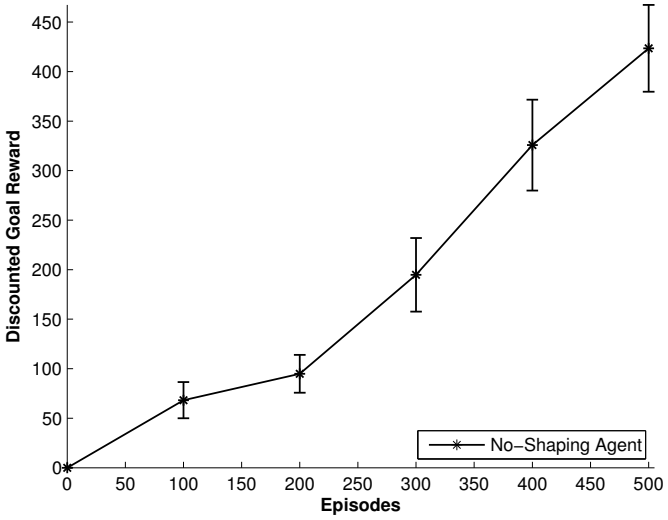

Fig. 3. Discounted reward for the RL agent without shaping, with error bars showing the standard error.

Figure 3 shows the averaged discounted, by γ, goal reward that the baseline agent achieved over the 30 experiments, including error bars showing the standard error.

Although the maximum discounted reward the agent can achieve cannot be directly computed, an estimate can be given to serve as a measure of performance. If we assume that no extra worker units are being built, then the total amount of minerals and gas the agent will need to collect are 1100 and 750 respectively as shown in Table I. When the agent performs a resource collection action, it gathers on average $\simeq 30$ resources. This amounts in a total of, including the construction actions, 69 actions to reach the goal. As a result, given that $\gamma = 0.99$ and the goal reward $= 1000$, then the expected discounted goal reward is $\simeq 500$.

It is apparent that the agent's performance is poor, compared to the estimated discounted reward it should achieve, with the agent only reaching a performance in the range of $300 \sim 400$. The complex nature of the chosen scenario slows down the learning process and inhibits any meaningful learning to occur within a reasonable time frame. The agent receives no guidance as to which states are considered desirable and as a result, spends most of its actions exploring, trying to form a good enough estimate of all state-action pairs in order to learn the optimal policy. It is worth noting that if the agent was let to run an infinite amount of time, it would eventually learn the optimal policy as the RL theory suggests. However, it would be impractical to run more than 500 episodes since even in this case, a single experiment was completed within $14 \sim 16$ hours even though the game settings were being set at the highest speed.

Extracting the behaviour of the agent showed that most of its actions were being spent performing redundant explorative moves especially at the start of the experiments. The agent was constructing extra buildings and units that did not contribute in reaching the goal and completing the task. There were many cases where the agent was spending a lot of its resources constructing extra buildings e.g. more than one Barracks, which resulted in the agent reaching the goal after having performed numerous unnecessary actions. There where also cases where the agent could not reach the goal, which is constructing the Battlecruiser unit, at all. As discussed previously, the agent starts with eighteen free slots which can be filled by creating worker or combat units. Looking at Table I, the SCV worker unit takes up one spot, while the Battlecruiser six. As a result the agent's exploration strategy, since no guidance was given, would many times lead to the creation of a large number of worker units, which take up all the free slots and therefore the Battlecruiser could never be built.

It is thus apparent that the complexity of this scenario inhibits learning within a practical amount of time, since the agent is exploring all possible states in order to acquire a good enough estimate, without receiving any guidance. The results clearly show that some form of guidance is necessary in order to affect the exploration strategy of the agent and guide it towards the goal. In the following sections we show how high level planning can be used to impact the low level behaviour of the agent in the form of plan-based reward shaping which was presented in Section III.

VII. Plan-based reward shaping design

As discussed previously, there are numerous occasions when abstract knowledge regarding a task can be provided to an agent in terms of a STRIPS plan to help guide exploration. In the case of this scenario, the knowledge can be that of the build order that the agent has to learn. While it is impossible to be exact regarding the number of worker units that need to be built, in order to optimise resource income and speed up the process of building the Battlecruiser, expert players know the correct build order that needs to be followed. This knowledge of the build order can be used as a STRIPS plan to efficiently guide the agent to the optimal policy.

```
BUILD(Barracks)
BUILD(Factory)
BUILD(Science Facility)
BUILD(Physics Lab)
BUILD(Starport)
BUILD(Control Tower)
BUILD(Battlecruiser)
```

Listing 1. Example Partial STRIPS Plan.

Given this domain, a partial example of the expected action-based STRIPS plan is given in Listing 1 and the full translated state-based plan used for shaping is given in Listing 2 with the $CurrentStepInPlan$ used by Equation 4 noted in the left hand column.

```
1        have(Barracks)
2        have(Factory, Barracks)
```

```
3       have(Science Facility, Factory,
        Barracks)
4       have(Science Facility, Physics Lab,
        Factory, Barracks)
5       have(Starport,Science Facility,
        Physics Lab, Factory,
        Barracks)
6       have(Starport, Control Tower,
        Science Facility, Physics Lab,
        Factory, Barracks)
7       have(Battlecruiser, Starport,
        Control Tower, Science Facility,
        Physics Lab, Factory,
        Barracks)
```

Listing 2. Full State-Based Plan.

VIII. Evaluation of the plan-based agent and comparison to the agent without shaping

In order to assess the performance of the plan-based agent against the agent using no shaping, the same experimental design was used as that presented in Section VI.

Similar to the agent without shaping, the agent implemented SARSA with $\epsilon-$greedy action selection and the agents' parameters were set such that $\alpha = 0.1$, $\gamma = 0.99$, and $\epsilon = 0.3$ linearly decreasing during an experiment.

These methods do not require the use of SARSA, $\epsilon-$greedy action selection or eligibility traces. Potential-based reward shaping has previously been proven with Q-learning, RMax and any action selection method that chooses actions based on relative difference and not absolute magnitude [18], and it has been shown before without eligibility traces [8], [3].

In all our experiments, we have set the scaling factor of Equation 4 to:

$$\omega = MaxReward/NumStepsInPlan \tag{5}$$

For environments with an unknown maximum reward the scaling factor ω can be set experimentally or based on the designer's confidence in the heuristic.

Each experiment lasted for 500 episodes i.e. games, and was repeated a total of 30 times. The agent using plan-based reward shaping is compared to the baseline RL agent without shaping. The agents are compared against the total discounted, by γ, reward they achieve. Note that the agents do not reach a reward of 1000, which is the reward for reaching the goal, as the reward they achieve is discounted by γ to account for the number of steps they performed, in order to reach the goal and finish the episode. The graphs include error bars which show the standard error. In addition, unpaired t-test were run with $p = 0.003$.

Figure 4 shows the performance of the agents comparing the discounted goal reward they achieve. The results are averaged over the 30 repetitions of the experiments. It is apparent that the agent using plan-based reward shaping manages to very early on in the experiment significantly ($p = 0.003$) outperform the baseline agent without reward shaping, and learn the optimal policy in the number of episodes that the experiment lasts.

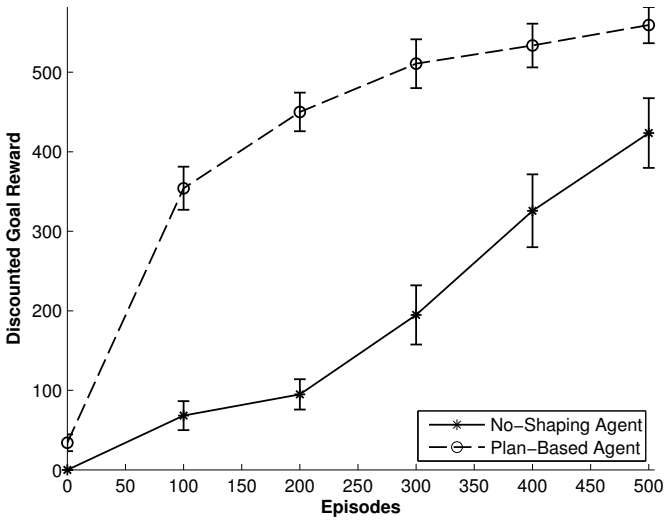

Fig. 4. Discounted reward for plan-based and no-shaping RL agents, with error bars showing the standard error.

The agent without reward shaping receives no guidance regarding states that seem promising and as a result spends most of its actions exploring in random locations of the state-space, until a good enough estimate is calculated. It is worth noting that if the agents were left to run an infinite amount of time, they would eventually reach the same policy as the RL theory suggests. However learning must be achieved within a practical time limit. Considering this scenario, running more than 500 episodes per experiments, so as to have the baseline agent learn the optimal policy, would result in the experiments taking months to complete since as stated previously, an experiment of 500 episodes was completed within $14 \sim 16$ hours even though the game settings were being set at the highest speed. This is not at all practical especially if RL needs to be incorporated into commercial games when we consider the tight deadlines the game industry works with.

The agent using plan-based reward shaping is not at all affected by the complex, large environment. Since it receives guidance regarding promising states, it takes less time to calculate which states are "good" and which are not. Even though the experiments last for only 500 episodes, it still manages to find the optimal policy. The agent reaches a performance in the range of $550 \sim 600$ which is higher than the expected discounted reward specified in Section VI. The reason is that the expected discounted reward was calculated under the assumption that no extra worker units are being built to facilitate the resource collection process. This is of course not optimal but only an estimate as a higher flow of incoming resources, means that the agent needs to spend less actions for resource collection. However, the optimal number of workers needed to built the Battlecruiser, and by extension the maximum discounted goal reward the agent should reach, cannot be known in advance and it is up to the agent to learn to optimise the resource collection process.

To better understand why the agent using plan-based reward shaping performs significantly better than the agent without shaping we have included a graph showing the average number of steps each agent performed, in order to find the goal

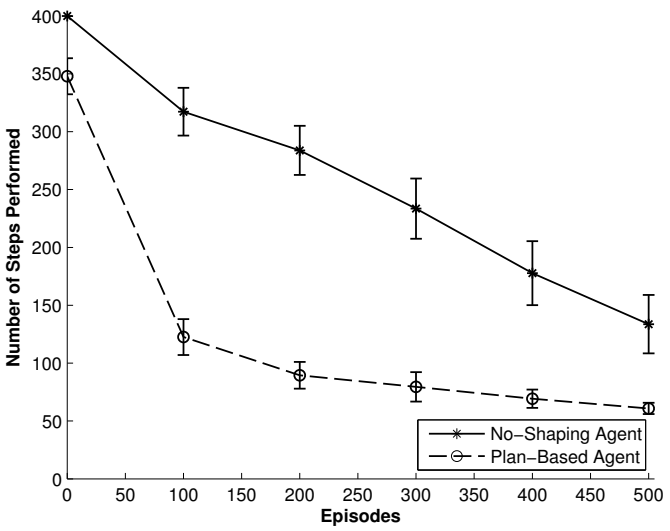

Fig. 5. Average number of steps taken by plan-based and no-shaping RL agents to complete an episode, with error bars showing the standard error.

and complete an episode, shown in Figure 5.

It is clear that the agent without reward shaping spends a lot of its actions exploring. Many actions that are irrelevant or unnecessary are performed resulting in the agent exploring states that do not provide any benefit to the learning process and are not a part of the optimal policy.

The agent using plan-based reward shaping manages to reduce the number of unnecessary steps very early on in the experiment. Guided towards promising states by the provided abstract knowledge, it only performs ≈ 50 steps within 200 episodes while the agent without reward shaping spends ≈ 200 steps at the end of the experiments.

IX. Discussion

After training the agents for the duration of the experiments some interesting conclusions can be drawn from their behaviour. The results show that the agent utilising plan-based reward shaping can learn the optimal policy within the 500 episodes of the experiments. An interesting behaviour was that of the worker units it built. The learning goal of the agent, as discussed previously, is to built the Battlecruiser unit. However, the agent also needs to utilise workers so that it can have a high enough flow of resources to built the Battlecruiser as fast as possible.

Given that the free slots for combat or worker units is eighteen, as discussed in Section V, one would expect that the agent would create twelve worker units which only take up one position in order to carry as much resources as possible, and then leave six slots for the Battlecruiser which takes up six positions as shown in Table I. However, extracting the agent's policy, showed that the agent instead created nine worker units to carry supplies. This shows that the agent not only managed to find the correct build order, but also managed to optimise the creation of worker units for gathering resources. This is particularly interesting since it shows that RL can be used in SC:BW to tackle multiple problems of the entire game one of which is resource management. It is also very encouraging especially if we consider combining RL agents in a hierarchical or hybrid manner, in order to create an RL player that would be able to play and efficiently manage the complexity of the entire game of StarCraft as the authors of [1] aim at.

In our experimental design we have tried to move towards more realistic and complex scenarios than that presented in [1], by having an agent that learns part of a strategy often employed by players in StarCraft. The results show that baseline RL cannot be used when scaling up to more complex tasks such as this. However, reward shaping, and in this case plan-based reward shaping, has allowed the agent to learn the optimal policy within a few hundred episodes.

We have managed to create an agent that can learn and deploy one specific strategy. We have also shown how the use of a high level planner can be used to compute a general strategy, that can guide the RL process by impacting the low level behaviour of the agent. However, there exist hundreds of different strategies in the game, some specific to races and others which are specific to certain situations. In the future we aim at creating agents that learn how to execute a multitude of different strategies that players of StarCraft often use. This agent could eventually be used as a module within a hierarchical or hybrid solution that deploys the learnt strategies according to specific situations. This will bring RL even closer to eventually having an agent that plays the entire game of SC:BW.

X. Conclusions

An attempt to evaluate RL in SC:BW was performed in [1] with very promising results. The chosen scenario however, was very limited and not representative of the complexity of the game in its entirety. We have chosen to move to a more realistic scenario which is the creation of a build-order manager, in order to learn part of a strategy often used by players, the *Terran Battlecruiser Rush* strategy.

As we move to more complex realistic scenarios, baseline RL techniques are no longer usable. We have demonstrated how a plan can be used in order to provide domain knowledge to an agent and also showed that it can help the agent tackle complex scenarios such as the one presented in this paper. We have shown how high level planning can be used to impact low level behaviour in the game of StarCraft. The results show that the agent utilising plan-based reward shaping is significantly better($p = 0.003$) in terms of discounted reward when compared to a baseline RL agent without reward shaping, as well as convergence time, which is very important especially if RL techniques are ever considered to become a part of commercial games.

StarCraft is a very complex environment but at the same time, given the large number of players, a lot of domain knowledge already exists that can be used to speed up the process of learning. We have decided to use plan-based reward shaping to create a build-order manager, but similar techniques can be used to tackle other problems inherent to SC:BW like resource management, building placement, scouting etc.

Our goal is to create a hierarchical or hybrid approach of a RL player that can efficiently manage the complexity of the entire game of SC:BW. We believe that our agent could

potentially be used as a module within a larger framework of agents, each tasked with managing different problems in SC:BW.

REFERENCES

[1] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game starcraft: Broodwar," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 402–408.

[2] M. Grześ and D. Kudenko, "Plan-based reward shaping for reinforcement learning," in *Proceedings of the 4th IEEE International Conference on Intelligent Systems (IS'08)*. IEEE, 2008, pp. 22–29.

[3] S. Devlin, M. Grześ, and D. Kudenko, "An empirical study of potential-based reward shaping and advice in complex, multi-agent systems," *Advances in Complex Systems*, 2011.

[4] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3, pp. 189–208, 1972.

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

[6] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY, USA: John Wiley and Sons, Inc., 1994.

[7] D. P. Bertsekas, *Dynamic Programming and Optimal Control (2 Vol Set)*. Athena Scientific, 3rd edition, 2007.

[8] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the 16th International Conference on Machine Learning*, 1999, pp. 278–287.

[9] J. Randløv and P. Alstrom, "Learning to drive a bicycle using reinforcement learning and shaping," in *Proceedings of the 15th International Conference on Machine Learning*, 1998, pp. 463–471.

[10] S. Devlin and D. Kudenko, "Dynamic potential-based reward shaping," in *Proceedings of The Eleventh Annual International Conference on Autonomous Agents and Multiagent Systems*, 2012.

[11] M. Grześ and D. Kudenko, "Multigrid Reinforcement Learning with Reward Shaping," *Artificial Neural Networks-ICANN 2008*, pp. 357–366, 2008.

[12] B. Marthi, "Automatic shaping and decomposition of reward functions," in *Proceedings of the 24th International Conference on Machine learning*. ACM, 2007, p. 608.

[13] G. J. Tesauro, "TD-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1994.

[14] I. Szita and A. Lörincz, "Learning tetris using the noisy cross-entropy method," *Neural computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[15] M. Smith, S. Lee-Urban, and H. Muñoz-Avila, "Retaliate: Learning winning policies in first-person shooter games," in *Proceedings of The National Conference on Artificia Intelligence*, vol. 22, no. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007, p. 1801.

[16] U. Jaidee and H. Muñoz-Avila, "Classq-l: A q-learning algorithm for adversarial real-time strategy games," in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[17] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram, "Transfer learning in real-time strategy games using hybrid cbr/rl," in *Proceedings of the twentieth international joint conference on artificial intelligence*, no. 1041-1046, 2007.

[18] J. Asmuth, M. Littman, and R. Zinkov, "Potential-based shaping in model-based reinforcement learning," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008, pp. 604–609.