Hierarchical Reinforcement Learning on the Virtual Battlefield

R. Tobi
0448710
rtobi@science.uva.nl

29^{th} of June, 2007
supervised by dr. Bram Bakker

Abstract

This paper investigates the potential of flat and hierarchical reinforcement learning (HRL) for solving problems within strategy games. A HRL method, Max-Q, is applied to a unit transportation task modelled within a simplified, discrete real-time strategy game engine, and its performance compared to that of flat Q-learning. It is shown that reinforcement learning approaches, and especially hierarchical reinforcement learning approaches, to strategy game AI are effective for learning such tasks. They are also efficient, in particular when such tasks decompose naturally into subtasks.

1. Introduction

Artificial Intelligence (AI) in computer games is a subject in which there is a lot of room for improvement. Most games employ a pre-scripted, reactive form of AI, which does not learn from experience. This is disadvantageous for several reasons. For one, once the player has become aware of the AI's behavior patterns, he can exploit them indefinitely, while the AI will be unable to offer any new challenges beyond that point. Secondly, because the AI will always solve problems (such as deciding where to attack) in an identical manner, it will frequently perform suboptimally even when the player is not yet capable of predicting all its decisions. Such flaws are typically simply masked by the game's designers by giving the AI extra advantages, rather than correcting the flaws themselves by incorporating more sophisticated AI techniques.

Reinforcement learning (RL) is an ideal candidate to address these shortcomings. RL is a subfield of machine learning, an area of artificial intelligence seeking to automate learning in computers. Techniques developed in reinforcement learning enable agents to adopt behavioral routines that were not explicitly pre-programmed by their developers. This is a particularly promising characteristic for so-called real-time strategy (RTS) computer games. RTS is a genre of computer wargames which take place in real-time, where resource gathering, base building, technology development and direct control over individual units are key components. The complexity of these games means there are a great many possible behaviors at any given moment and choosing the best is non-trivial even for a human player. Programming an AI agent to play an RTS game well is thus a very difficult task, one that could be made easier via the use of reinforcement learning methods. However, while some initial work ([4], [5]) has been done to enhance game AI this way, outside of academic circles these methods largely continue to be ignored.

In this paper the power of reinforcement learning to solve a specific subproblem common to the RTS genre is explored, that of transporting units across the battlefield in the presence of enemy forces. A transportation scenario has been modelled within a custom-written, discrete RTS game engine, and two RL algorithms, Q-learning and MAXQ-0, are evaluated on this scenario. The MAXQ-0 algorithm merits special attention in that it is hierarchical rather than flat. Flat RL represents and learns a task as one monolithic problem at one level of abstraction, with one set of states and one set of primitive actions. Recent advances in RL [10] have produced a number of approaches to combat the curse of dimensionality (the exponentional growth of the number of parameters that must be learned as the state becomes more complex) that traditional flat methods are faced with.

One of these approaches consists of abstracting over time, where decisions correspond not to one-step but to temporally-extended actions (subtasks). Temporal abstraction in turn leads to hierarchical control frameworks and learning algorithms, of which MAXQ-0 is an example. The performance of both algorithms is demonstrated to be convincing grounds for improving game AI with them.

The rest of this paper is layed out as follows. Section 2 gives an overview of reinforcement learning theory and the MAXQ hierarchical RL formalism. Section 3 details the application of the theory to the transportation problem. In section 4 the experimental results are presented. Finally, sections 5 and 6 make concluding remarks and discuss possible issues, while section 7 offers some suggestions for extending the work described here.

2. Theoretical Background

Reinforcement Learning

Reinforcement learning is a computational approach to learning from experience by interaction with an initially unknown, stochastic, but fully observable environment. Compared to other machine learning approaches, it is much more goal-directed. RL is not a supervised learning method like artifical neural networks and decision trees. In supervised learning the designer of the learning task provides the agent with a number of training examples, and the agent's success is based on how well it generalizes from these training examples. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

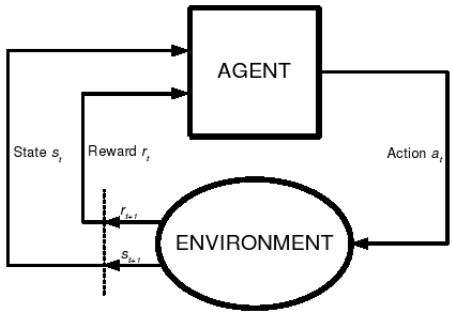

Figure 1: The Reinforcement Learning Model

Figure 1 shows the RL framework. An agent interacts with the environment by selecting an action in each state, which places the agent in a new state and produces a numerical reward signal from the environment. The agent is

thus not told whether its actions are good or bad, only what they are worth.

More formally, the agent has a representation of the state of the environment, $s_t \in S$, at each discrete time step $t = 0, 1, 2, \ldots$, where S is the set of possible states. The agent selects an action $a_t \in A(s_t)$, on the basis of its current state information, where $A(s_t)$ is the set of possible actions in state s_t. In doing so the agent receives a numerical reward $r_{t+1} \in R$, and ends up in a new state s_{t+1}. RL methods learn by experience a policy π_t, a mapping from states to action selection probabilities (or directly to actions if the policy is deterministic). $\pi_t(s, a)$ is the probability for selecting action $a_t = a$ if $s_t = s$. The general goal of reinforcement learning is to arrive at a policy that maximizes the expected future cumulative reward.

The expected future cumulative reward is also called the expected return, denoted R_t. In cases where the learning process can be broken down into subsequences (episodes) and where there exists a final time step T for each episode (which corresponds to the agent being in a terminal state), the expected return is defined as

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T = \sum_{k=0}^{T} r_{t+k+1} \tag{1}$$

where $r_{t+1} + r_{t+2} + \cdots + r_T$ is the sequence of rewards received after timestep t. Such tasks are called episodic. A typical example of an episodic task is that of playing a game, which terminates when one round of the game is finished. Many tasks however are not episodic, and can not naturally be broken down into episodes that have terminal states. These tasks are called continuous. For continuous tasks $T = \infty$, and therefore, if R_t were defined as in equation 1, the expected return would be infinite. The concept of *discounting* is introduced to avoid this. The goal of the agent in a continuous task is to maximize the expected discounted return

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

where γ is the discount rate parameter, $0 \leq \gamma \leq 1$. If γ is zero then the agent only tries to maximize the immediate reward, whereas as γ approaches 1 the agent becomes more far-sighted, meaning that future rewards are more strongly weighed in the calculation of the discounted return. Equations 1 and 2 can be expressed in a unified notation by

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{3}$$

where $T = \infty$ for continual tasks and $\gamma = 1$ for episodic tasks. The reward function is the designers tool for telling the agent what to achieve. It is

therefore very important to design this function carefully, so that the reward signals for the various states and actions really represent the goal the designer wants the agent to fulfill. The problem of assigning the appropriate rewards is known as the credit assignment problem.

Markov Decision Processes

The mathematical foundations for reinforcement learning rely on the assumption that the environment has the *Markov property*. In this case the learning task can be modelled as a Markov Decision Process (MDP). RL problems are said to have the Markov propery if in all situations the state signal received by the agent from the environment contains all relevant information about the environment. This means that the agent can fully observe the environment at any given time. Chess is an example of a problem that satisfies the Markov property. If a problem has the Markov property, the new state s_{t+1} only depends on the current one s_t and the action a_t chosen by the agent.

Formally, in the general case the environment's response at time t depends on the sequence of all earlier events $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$ expressed as the probability distribution in equation 4. However, if a problem has the Markov property, the response only depends on the current state s_t and the action a_t as expressed by equation 5. A problem satisfies the Markov property (and is thus an MDP) if and only if equation 4 is equal to equation 5

$$P\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, \ldots, r_1, s_0, a_0\} \tag{4}$$

$$P\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \tag{5}$$

for all s', r, and all possible earlier events $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$. More specifically, if the state and action spaces are finite, the RL task is a *finite* MDP. A finite MDP is completely defined by the state space, action space, and the one-step dynamics of the environment

$$P(s' \mid s, a) = P\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{6}$$

$$R(s' \mid s, a) = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{7}$$

where $P(s' \mid s, a)$ is the probability distribution that the next state is s' given state s and action a, and $R(s' \mid s, a)$ is the expected value of the next reward given state s, action a and next state s'.

Certainly not all RL tasks satisfy the Markov property, and many RL algorithms do not assume a perfect model of the environment's dynamics. For such tasks the state signal from the environment can be thought of as an approximation of the the Markov property, where the policy and value functions are functions of the current state and the selected action only.

The Value Function

In order to learn how to maximize the discounted return, most RL algorithms estimates value functions. These value functions give the expected return for the agent for being in a state or for performing an action in a state according to the current policy π^t. The state-value function for a policy π, denoted $V^\pi(s)$, is the value of starting in state s and thereafter following policy π. It has the definition

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P(s' \mid s,a)[R(s' \mid s,a) + \gamma V^\pi(s')] \end{aligned} \tag{8}$$

where E_π is the expected value given that the agent follows policy π and the next state is s'. Equation 8 is called the Bellman equation for V^π and says that the value function can be recursively defined as a relationship between the value of the current state s and that of the next state s'.

Closely related to the value function is the action-value function for a policy π, denoted $Q^\pi(s,a)$. This is the expected return for starting in state s, taking action a, and thereafter following policy π. It has the definition

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t \mid s_t = s, a_t = a\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \\ &= \sum_{s'} P(s' \mid s,a)[R(s' \mid s,a) + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')] \end{aligned} \tag{9}$$

where equation 9 is the corresponding Bellman equation for Q^π.

The value functions are bound to a policy. In maximizing the value functions, reinforcement learning algorithms thus try to find the optimal policy for solving the task. This optimal policy is denoted π^*. A policy for a finite MDP is defined to be optimal if and only if $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and all policies π. All optimal policies share the same state-value function. The value function for the optimal policy is the optimal state-value function

$V^*(s)$

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ &= \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)[R(s' \mid s, a) + \gamma V^*(s')] \end{aligned} \tag{10}$$

for all $s \in S$. The optimal policies also share the same optimal action-value function, defined as

$$\begin{aligned} Q^*(s, a) &= \max_\pi Q^\pi(s, a) \\ &= \sum_{s'} P(s' \mid s, a)[R(s' \mid s, a) + \gamma \max_{a'} Q^*(s', a')] \end{aligned} \tag{11}$$

for all $s \in S$ and all $a \in A$. Reaching the optimal policy is an interleaved iterative process of policy evaluation and policy improvement.

Exploration vs. Exploitation

An important issue in RL is the so-called exploration-exploitation problem faced by the agent during learning. For every state there is always at least one action whose estimated action-value is the largest, the greedy action $a = \arg\max_{a \in A(s_t)} Q(s_t, a)$. If the agent chooses this greedy action, it *exploits* its current knownledge of the environment. This is frequently a good thing to do, especially if the learning process has converged to the optimal value function or if the agent wants to maximize his immediate reward. In the early stages of the learning process however, the agent has to *explore* the environment by taking non-greedy actions. By exploring the environment the agent gains new knowledge and improves its estimate of the value function.

To leave the door open for exploration, the ϵ-greedy action-selection method is a simple alternative to the purely greedy action selection scheme. In ϵ-greedy selection, t the agent selects the greedy action most of the time, but with small probability ϵ takes a random one. This ensures that, in the limit, the agent will discover the optimal value function. However, a drawback of this method is that the agent selects all non-greedy actions with equal probability. An alternative is to rank the actions by their estimated action-value functions for the current state. This is called *softmax* action selection and the most common softmax method uses a Boltzmann distribution. For softmax selection, the selection probability for an action a_t at timestep t is defined as

$$P(a_t) = \frac{e^{Q_{t-1}(s_t, a_t)/\tau}}{\sum_{b \in A(s_t)} e^{Q_{t-1}(s_t, b)}/\tau} \tag{12}$$

where the parameter τ is called the temperature. As τ approaches infinity the action selection becomes more random, while as τ approaches zero it

becomes greedier. The right balance between exploration and exploitation depends strongly on the RL task and setting the right τ and ϵ values is therefore very important for obtaining good results. A common strategy for the ϵ-greedy method is to initialize ϵ to a very high value (so that the agent explores a great deal) and then to gradually decrease it over time to promote more greedy behavior.

Temporal-Difference Learning

Three fundamental classes of algorithms can be distinguished in reinforcement learning: Dynamic Programming (DP) methods, Monte Carlo (MC) methods and Temporal Difference (TD) methods. DP methods are algorithms that can calculate the optimal policy, provided that a full model of the environment's dynamics is known. The big drawback of DP methods are that they are computationally very expensive. Monte Carlo algorithms do not need a perfect model of the environment, but are all based on averaging sample returns. They are therefore only applicable to episodic tasks, because good return estimates are only available after completing an episode. For the learning task documented in this paper only TD methods have been used (Q-learning and $MAXQ$-0), so the others are not covered in detail.

TD methods have the following properties:

- They do not require a complete model of the environment.
- They estimate the state-value (or action-value) function based on earlier estimates of the value functions, ie. they do not wait until an episode is finished before updating the function estimates, but do this every timestep. This concept is called bootstrapping.
- They are naturally implemented in an online manner, mixing computation and action from sample or simulated experience.

The central part of TD learning is the *temporal-difference error*, δ_t, which is defined as

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{13}$$

The TD error represents the difference between two temporally successive value predictions, and is used to move the value function closer to the optimal one. A positive TD error indicates that the action-value for selecting action a_t in state s_t, $Q(s_t, a_t)$ should become larger, while a negative TD error indicates it should become smaller.

Watkin's Q-learning [6] is perhaps the most famous TD reinforcement learning algorithm. In Q-learning the agent estimates the optimal action- value

function Q^* directly, instead of estimating the state-value function V^π or the action-value function Q^π. The update rule for Q-learning is given by

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

where α is a step-size parameter called the *learning rate*, $0 \leq \alpha \leq 1$. At each timestep the value function estimate is updated by the TD error multiplied by the learning rate. The learning rate determines the proportion of the TD error that is used in the update. In general, low values for α yield slower but more stable learning than high ones. Furthermore, by taking the maximum over all actions a of the Q-values for the next state s_{t+1}, Q-learning directly approximates the optimal action-value function independent of the actual policy being followed by the agent. In other words, it is an off-policy algorithm. It has been proven that Q-learning converges with probability 1 to Q^* under the constraint that in the limit, each state-action pair is visited infinitely often. Pseudo-code for Q-learning is given in [3].

Hierarchical Reinforcement Learning

As mentioned previously, RL methods suffer from the curse of dimensionality: learning times increase exponentially with the size of the state-space (or the state-action space), which is treated as one large "flat" table with one entry per state or state-action pair. In recent years much work has gone into combating this problem by introducing various kinds of abstraction to the RL framework. Examples of this are the HAM (Hierarchy of Abstract Machines) method of Parr and Russell [7] the "Option" method of Sutton et all [8], as well as many types of function approximation methods [3]. Through such abstraction, hierarchical reinforcement learning (HRL) seeks to find hierarchical structures in complex MDP's. This is done by breaking down the overall learning task (MDP) into smaller sub-MDP's, which creates a task hierarchy. The actions of each subtask in the hierarchy are then either other subtasks, or primitive commands which directly influence the agent's state.

Many of the subtasks in the task hierarchy represent abstract subgoals that often can not be accomplished in one time step. A subtask is therefore extended in time and the subtask is active until a termination condition is fulfilled. A termination condition is either fulfilled because the goal of the subtask has been completed or because the subtask is not applicable in the current state. The subtasks each learn their own policies for a subset of the state space S.

Because actions can now take a variable amount of time to complete, the natural mathematical foundation for HRL is the semi-Markov Decision Process (SMDP). This is a generalization of the MDP model which supports

temporally extended actions. For discrete SMDP's the state transition probability function is extended to a joint distribution of the next state s' and the number of discrete timesteps N given that action a was executed in state s:

$$P(s', N \mid s, a) = P\{s_{t+1} = s', N \mid s_t = s, a_t = a\} \tag{14}$$

The expected reward is also extended to depend on the number of timesteps, as follows:

$$R(s', N \mid s, a) = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s', N\} \tag{15}$$

With these two definitions the semi-MDP versions of the Bellman equations become

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s', N} P(s', N \mid s, a)[R(s', N \mid s, a) + \gamma^N V^{\pi}(s')] \tag{16}$$

$$Q^{\pi}(s, a) = \sum_{s', N} P(s', N \mid s, a) \left[R(s', N \mid s, a) + \gamma^N \sum_{a'} \pi(s', a') Q^{\pi}(s', a') \right] \tag{17}$$

Note that the discount factor γ is now decreased exponentially by the number of timesteps N taken to complete action a.

MAXQ

MAXQ is a HRL framework by Dietterich [1]. It is accompanied by two online, model-free learning algorithms, MAXQ-0 and MAXQ-Q. If the designer can decompose an overall learning task into distinct subtasks, the MAXQ framework can be used to build a MAXQ graph consisting of two different types of subtasks: primitive subtasks (leaf nodes) that execute commands for the agent, and composite subtasks (inner nodes) that select other subtasks to solve their tasks. The MAXQ graph recursively describes how to decompose the overall value function for a policy into a collection of value functions for the subtasks.

In formal terms a MAXQ decomposition decomposes an MDP M into a set of subtasks $\{M_0, M_1, \ldots, M_n\}$, where M_0 is the root MDP representing the whole learning problem. A subtask in turn is a three-tuple $<T_i, A_i, \tilde{R}_i>$, where

- $T_i(s_i)$ is a termination predicate that divides S into a set of active states S_i and a set of terminal states T_i. A subtask M_i can only be executed if the current state s_i is in S_i.
- A_i is the set of actions that can be performed to achieve subtask M_i. These actions can either be primitive $\in A$ (the set of primitive actions for M), or they can be other subtasks that are children of M_i.

- $\tilde{R}_i(s' \mid s, a)$ is the pseudo-reward function which specifies a pseudo-reward for each transition from a state $s \in S_i$ to a terminal state $s' \in T_i$. This function tells how desirable each of the terminal states is for the subtask. Pseudo-reward are used in the MAXQ-Q algorithm to speed up the learning of subtasks and to overcome some problems associated with recursive optimality (see below). Typically, goal terminal states have a pseudo-reward of 0 and all non-goal terminal states have negative pseudo-rewards. For MAXQ-0 the pseudo-reward function $\tilde{R}$ is always zero.

Primitive actions are primitive subtasks or leaf-nodes in the MAXQ decomposition. These are always executable, always terminate immediately, and have no pseudo-reward.

In MAXQ subtasks have their own policies. The collection of policies is called a hierarchical policy. A hierarchical policy π is thus a set containing a policy for each of the subtasks: $\pi = \{\pi_0, \ldots, \pi_n\}$. In a hierarchical policy each subroutine (subtask policy) executes until it enters a terminal state $\in T_i$ for its subtask i. The MAXQ method discovers a *recursively optimal policy* for an MDP M (decomposed as $\{M_0, \ldots, M_k\}$), which is a hierarchical policy $\pi = \{\pi_0, \ldots, \pi_k\}$ such that for each subtask M_i, the corresponding policy π_i is optimal for the SMDP defined by the state-set S_i, the action-set A_i, the state transition probability function $P^\pi(s', N \mid s, a)$ and the reward function that is the additive combination (ie. the sum) of $R(s' \mid s, a)$ and $\tilde{R}(s')$.

Recursive optimality is a form of local optimality in which the policy at each node of the MAXQ graph is only optimal given the policies of its children, recursively. This follows from the definition of the state transition probability distribution. Each subtask attempts to discover an optimal policy without reference to the policy of its parent. It is this feature that enables subtasks to be shared and re-used for multiple parent tasks, accelerating learning. In addition, the context-free nature of the individual value functions creates opportunities for state abstraction, which can be seized to further enhance learning [11].

A visual example of a recursively optimal policy is shown in figure 2 below. The subtask policy for exiting the left room is optimal, but the global policy for reaching the goal state G from within the left room is not. In MAXQ-Q, pseudo-rewards can be used to alleviate this problem.

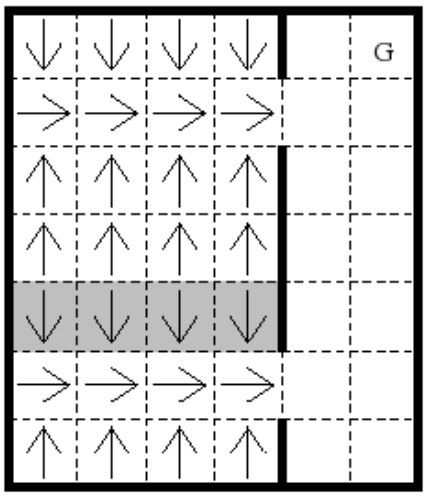

Figure 2: A Recursively Optimal Policy

The value function $V\pi(s)$ for a hierarchical policy π is known as the *projected value function*. The projected value function is the value $V^\pi(i,s)$ for executing π starting in state s at the root of the task hierarchy. For each task i and state s in the hierarchy this value is decomposed into two parts

$$V^\pi(i,s) = \begin{cases} Q^\pi(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s' \mid s,i) R(s' \mid s,i) & \text{if } i \text{ is primitive} \end{cases} \tag{18}$$

where R is the reward function for the primitive subtasks and $Q^\pi(i,s,a)$ is recursively defined as

$$Q^\pi(i,s,a) = V^\pi(a,s) + C^\pi(i,s,a) \tag{19}$$

$C^\pi(i,s,a)$ is called the completion function, which represents the discounted cumulative reward for completing subtask M_i after invoking the subroutine (policy) for subtask M_a in state s:

$$C^\pi(i,s,a) = \sum_{s',N} P_i^\pi(s', N \mid s,a) \gamma^N Q^\pi(i,s',\pi(s')) \tag{20}$$

Equations 19, 20, and 21 together form the *decomposition equations* for the task hierarchy [1]. They prescribe how to decompose the projected value function for the root, $V^\pi(0,s)$, into the projected value functions for each subtask and the individual completion functions $C^\pi(j,s,a)$ for $j = 1,\ldots,n$. The projected value function is stored in a distributed fashion: as V values for the primitive subtasks in the hierarchy and as C values for the composite subtasks. Figure 3 depicts the decomposition; the sequence

$r_1, \ldots, r_14$ denotes the rewards received from primitive actions at timesteps $1, \ldots, 14$. The MAXQ value function decomposition has the general form

$$V^\pi(0, s) = V^\pi(a_m, s) + C^\pi(a_{m-1}, s, a_m) + \cdots + C^\pi(a_1, s, a_2) + C^\pi(0, s, a_1) \tag{21}$$

where $a_0, a_1, \ldots, a_m$ is the chain of subtasks chosen by the hierarchical policy from the root down to a leaf-node.

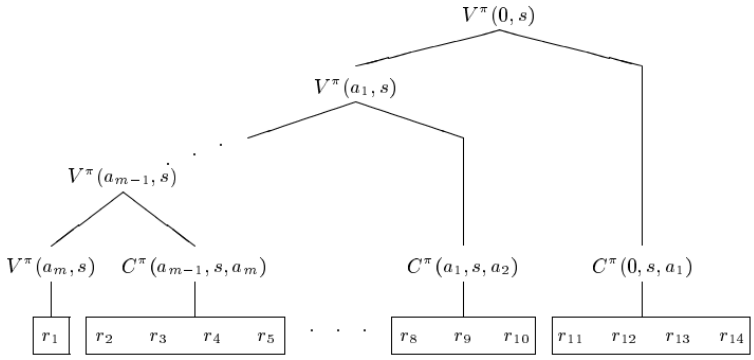

Figure 3: The MAXQ Value Function Decomposition

The MAXQ task decomposition is graphically represented by a MAXQ graph. This graph contains two types of nodes, Max-nodes and Q-nodes. The Max-nodes represent the subtasks in the task decomposition. The Q-nodes represent the actions that are available for each subtask. The Max-nodes "compute" the projected value function $V^\pi(i, s)$ for subtask i, by retrieving from their Q-node children the value of $Q^\pi(i, s, a)$. The Q-nodes themselves calculate these values by retrieving from their children a the projected values of $V^\pi(a, s)$ and then adding their completion function values $C^\pi(i, s, a)$. An an example, figure 4 shows the graph for the task discussed in the next section of the paper.

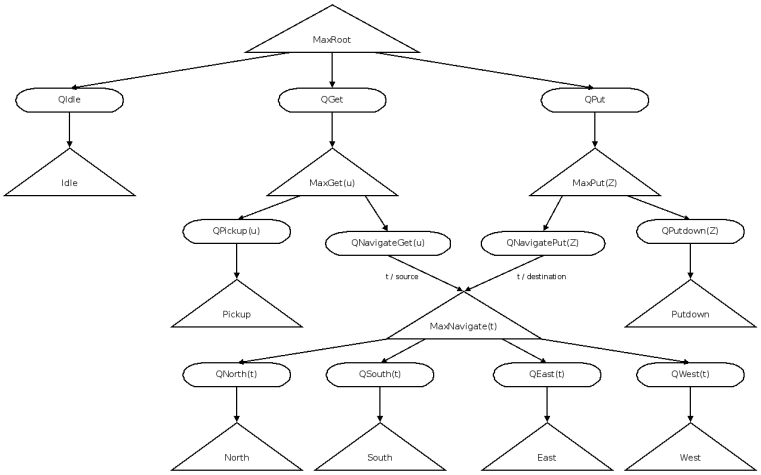

Figure 4: The MAXQ Graph for the Unit Transportation Task

Some of the nodes in this graph are parameterized. During learning these parameters are bound (instantiated) to an argument that makes the subtask concrete. For instance, the parameter t in the *MaxNavigate* task can be bound to any element of the set $\{R, G, B, Y, Z\}$, which delineates five concrete navigation subtasks.

MAXQ-0

Two learning algorithms accompany the MAXQ method, MAXQ-0 and MAXQ-Q. Only MAXQ-0 has been implemented for the research conducted in this paper, so a description of MAXQ-Q is not provided here.

MAXQ-0 is a recursive function that executes the current exploration policy π_x starting at Max-node i and in state s. It performs actions until it reaches a terminal state for MDP M_i. After returning from a recursive call to execute an action $\pi_x(i, s)$, it updates the completion function value for node i. If the learning task is discounted, the returned count of the number of primitive actions is used to discount the value of the state s' that resulted from this execution. At leaf nodes the algorithm updates the estimated one-step expected reward $V(i, s)$, with a learning rate α_i that can vary per node but should be decreased to zero over time. The function *SelectNextTask(MaxNode, State)* selects an action according to an ordered GLIE (Greedy in the Limit with Infinite Exploration) policy based on the actions' Q-values. This policy is required to have a fixed order so that MAXQ-0 converges to a uniquely defined recursively optimal policy [2].

Pseudocode for MAXQ-0 is given below. Note that the learning rule for updating the completion function on line 13 is a kind of Q-learning for

SMDP's, and the learning of the projected value function for the primitive actions on line 4 is accomplished by Q-learning where the discount factor is set to zero. This means that the primitive actions only try to maximize the immediate reward and do not perform any value prediction.

Table 1: The MAXQ-0 Learning Algorithm

```
1  function MAXQ0(MaxNode i, State s)
2    if i is a primitive MaxNode then
3      execute i, receive r, save new state s'
4      V(i,s) := (1 − α_i) · V(i,s) + α_i · r
5      return 1
6    else
7      actionCount := 0
8      while i not is terminal do
9        a := SelectNextTask(i, s)
10       numActions := MAXQ0(a, s)
11       retrieve saved state s'
12       v := γ^numActions · EvalMaxNode(i, s')
13       C(i,s,a) := (1 − α_i) · C(i,s,a) + α_i · v
14       actionCount := actionCount + numActions
15       s := s'
16     return actionCount

17 function EvalMaxNode(MaxNode i, State s)
18   if i is a primitive MaxNode then
19     return V(i,s)
20   else
21     sum := −∞
22     for each MaxNode j in Actions_i do
23       v := EvalMaxNode(j, s)
24       c := C(i,s,j)
25       if (v + c) > sum then
26         sum := v + c
27     return sum
```

3. Application

To bring the theory to bear on RTS games, implementations of Q-learning and MAXQ-0 were written and linked to a small strategy game engine. Within this engine a decision problem was set up that can be seen as a miniature reflection of a central task faced by AI in strategy games, the

transportation of units across enemy-occupied terrain. This task will now be explained in detail.

The Unit Transportation Task

In the Unit Transportation Task (UTT) a transport unit (depicted by a triangle) is randomly placed in a 5 by 7 gridworld to the left of an enemy that patrols up and down the center column. Two other units (which do not move) are spawned on one of the four special cells labeled R, G, B, and Y. It is the transport's job to transfer these units to the cell labeled Z while avoiding getting too close to the enemy unit. The world is schematically shown in figure 5. An episode in this world ends when both units are at Z.

In flat Q-learning, the transport can choose one of the following one-step actions at each timestep: Up, Down, Left, Right, Get, Put, and Idle. All actions are deterministic. There is a reward of -1 for each move action, $+1$ for a (legal) Get, $+20$ for a (legal) Put, and -1 for an Idle action. If a Get or Put action is attempted illegally, the reward is -10. Illegal move actions are no-ops and yield the same reward as legal ones. If the transport is carrying both units when it executes a legal Put, the reward is doubled to $+40$. Additionally, the transport receives a penalty of $\frac{-20}{d+1}$ for every action, whether legal or illegal, that is executed within a squared Euclidean distance $d \leq 4$ from the enemy. If the squared distance is greater than 4, then this penalty is zero.

The MAXQ-0 decomposition for the UTT has already been shown in figure 4. The *MaxGet* and *MaxPut* tasks are responsible for picking up and dropping off one of the two units. Again the reward for the primitive Put operation is doubled if it is legal (that is, if the transport is at Z) and both units have been picked up. The *MaxNavigate* task serves to move the transport to one of the five labeled cells. The leaf Max-nodes contain the same primitive actions that can be selected by flat Q-learning, and produce the same one-step rewards. *MaxGet*, *MaxPut*, and *MaxNavigate* are parameterized to keep the task-graph compact. These subtasks also have parameterized termination predicates, which are key to this reduction.

For both algorithms a state is represented by a four-tuple $<U_1, U_2, P, E>$, where

- U_1 is the location of unit 1, $U_1 \in \{R, G, B, Y, Z, T\}$.
- U_2 is the location of unit 2, $U_2 \in \{R, G, B, Y, Z, T\}$.
- P is a coordinate-pair <y, x> that indicates the transport's position on the grid.

- E is a triple (y, x, h) that indicates the enemy unit's position on the grid and its direction of movement, $h \in \{-1, 1\}$.

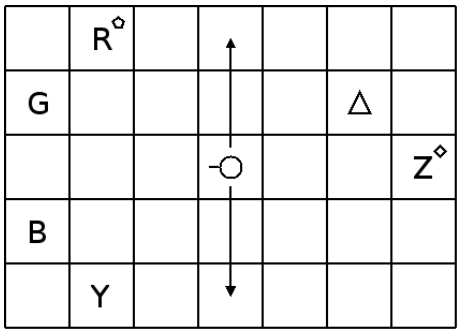

Figure 5: The RTS Game Environment

The total number of states is therefore 12.600. This task illustrates the importance of temporal abstraction and sharing subtasks. Navigating the transport is a temporally extended activity that can last several timesteps, while also being an element of both the Get and the Put subtasks. If the AI learned how best to navigate to each possible destination, then Get and Put could share the solution to this task directly. Q-learning by contrast would have to try all actions in all states to learn that in most it should navigate rather than attempt eg. a pickup.

4. Results

To generate results, the algorithms were run in 10 series of 10.000 episodes each (a *trial*) and their output averaged over all series. Both Q-learning and MAXQ-0 select actions by means of the ϵ-greedy scheme with variable ϵ. After each episode ends, ϵ is geometrically decayed by a factor β until it has reached the value 0.01, at which point the decay stops.

The averaging process was done for a number of parameter settings combinations. For the first combination of settings, ϵ was initialized to 0.333 and β to 0.999 for both Q-learning and MAXQ-0. The UTT is undiscounted, so γ was left at 1. V- and C- values were initialized to 0.1234 and 0.5678 respectively. During all experiments, learning rates were kept constant at 0.25 at all nodes in MAXQ-0 and in Q-learning's update rule. Figure 6 shows the mean output of a trial using the first settings combination. On the x-axis the average total number of primitive actions (timesteps) executed during a trial is plotted. The y-axis shows the cumulative reward per episode. Since episodes take less and less actions to complete as the algorithms converge, the reward per episode increases as a series progresses.

MAXQ-0 outperforms Q-learning in this trial by a wide margin, converging at around 100.000 primitive actions while Q-learning needs roughly twice as much time to do so. It is also evident that the task decomposition is beneficial through the reduced number of policies considered for each task. Q-learning starts out six times lower because of the large negative rewards for illegal *Get* and *Put* actions, which MAXQ-0 learns to avoid quickly.

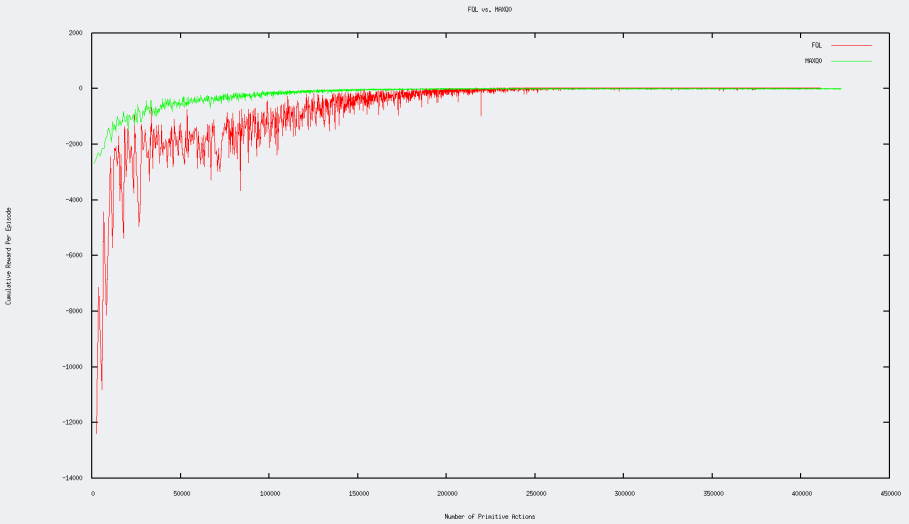

Figure 6: FQL vs. MAXQ0 (1), $\beta = 0.999$

Two other combinations of settings were tested, differing in the ϵ values for both algorithms. In none of these two additional trials did Q-learning come close to MAXQ-0's level of performance.

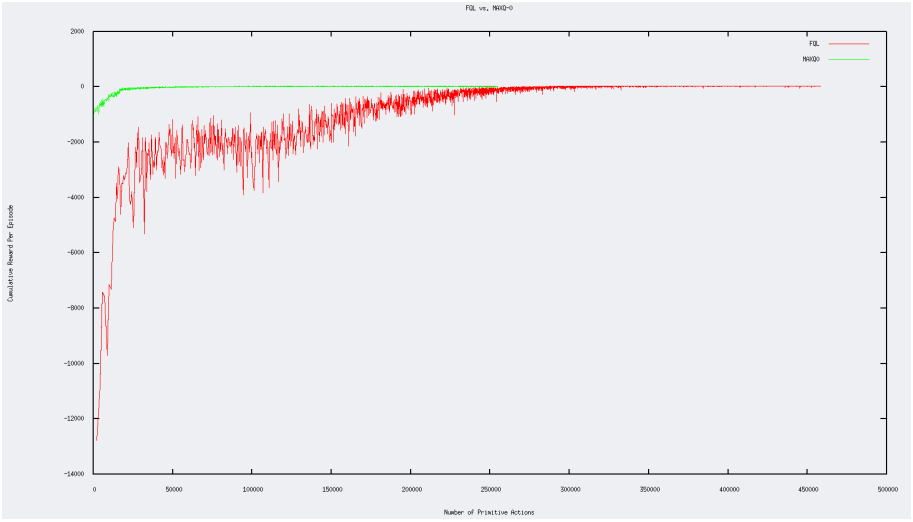

Figure 7: FQL vs. MAXQ0 (2), $\beta = 0.999$

The second trial, $\epsilon_{FQL} = 0.666$ and $\epsilon_{MAXQ0} = 0.111$.

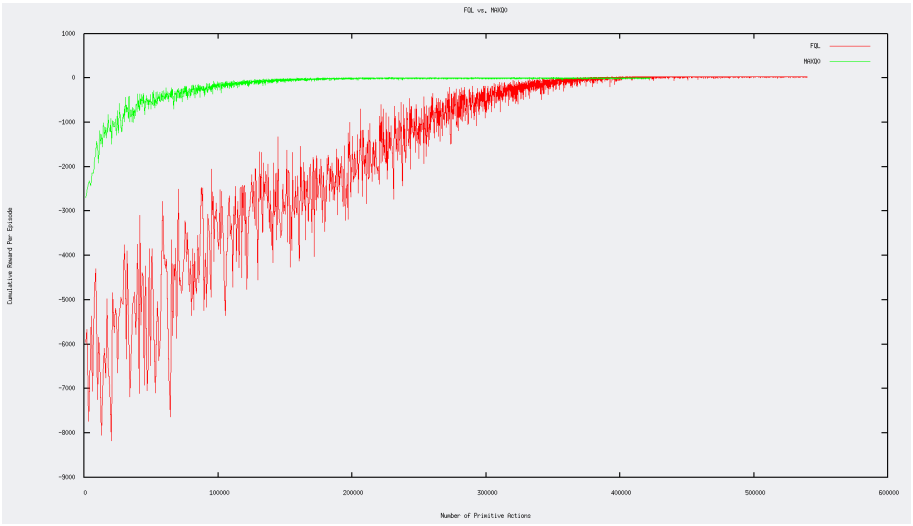

Figure 8: FQL vs. MAXQ0 (3), $\beta = 0.999$

The third trial, $\epsilon_{FQL} = 0.999$ and $\epsilon_{MAXQ0} = 0.333$.

5. Conclusion

In this paper two principle reinforcement learning algorithms have been pitted against each other on an hierarchical RTS game AI subtask modelled

inside of a small custom-built game engine. Flat Q-learning has been empirically demonstrated to be inferior to MAXQ-0 in terms of convergence rate even without exploiting the possibilities for state abstraction offered by the MAXQ decomposition. Both are good candidates to function as game AI modules, but MAXQ-0 offers greater scaling potential. The higher performance of MAXQ-0 does come at the price of increased implementation difficulty, computational cost per timestep, and requires the programmer to specify the task hierarchy in advance.

6. Discussion

The results presented in this paper make a convincing case for reinforcement learning as a vehicle to augment AI in strategy games. However, it is worth reflecting on some issues that can arise in a practical context and how they might be dealt with.

One reason why RL methods have so far not been applied to games is the lack of control they present to their designers. Often a game is designed in a pre-scripted way to provide the player with a carefully arranged linear path through its storyline. If on the other hand the game's AI acted without supervision, it could change the player's experience into something different than intended by the designers. Another reason is that the purpose of AI should be to entertain the player, not to make the game so difficult that it is no longer enjoyable to play. The performance of Q-learning or MAXQ-0 would therefore have to be restricted to keep the player from becoming frustrated and giving up. These points imply that, at least for a game's single-player portions, the RL routines might in some cases lead to undesirable effects. However, for game modes that do not depend on the storyline being followed, they would still be preferable to scripted AI.

A major disadvantage of the RL framework is that it assumes a fully observable environment. In RTS games this assumption is almost always violated due to the presence of a simulated fog of war, which introduces uncertainty about the exact state of the battlefield (eg., the location of enemy forces). In principle this would render both Q-learning and MAXQ-0 unsuitable for RTS game AI unless, for instance, theory from partially-observable MDP's [9] was incorporated into them. This however would substantially increase their complexity, and would moreover only function up to a certain degree. A better approach might be to allow all of the game-state information to be accessible by the AI, so that no adjustment would be required to either algorithm other than the appropriate state discretizations.

A final issue concerns that of speed. Full-fledged RTS games are saddled with

a vast state-action space that is likely beyond the ability of even MAXQ-0 combined with safe-state abstraction to explore in any reasonable amount of time. Of course, for solving subproblems such as the transportation task outlined in this paper, much of this space is actually irrelevant, and state abstraction could resolve this issue to a significant degree [11]. Nevertheless solutions should be sought if RL algorithms such as MAXQ-0 are to gain a foothold in game AI, perhaps in the area of distributed computation.

7. Future Work

Several directions could be taken to expand upon the work presented in this paper. The environment could be enlarged and filled with more enemy forces and units to transport to different locations, and more tasks (such as Refuel and Repair) could be assigned to the transport itself. This would better tax the subtask sharing effectiveness of MAXQ-0 and widen the gap between it and Q-learing. The state transitions could be made probabilistic for certain actions (eg. the moves) to further challenge MAXQ-0's learning capabilities. Naturally the task itself need not be restricted to one of transportation either. An RTS game AI must tackle many tasks in parallel, so the MAXQ formalism could well be used to decompose other, higher-level activities, eg. deciding where and which targets to attack. Finally, for larger hierarchies with many terminal states per node, the integration of pseudo-rewards to guide the recursively optimal policy might be a logical next step.

References

[1] T. G. Dietterich: *Hierarchical Reinforcement Learning with the MaxQ Value Function Decomposition.* Journal of Artificial Intelligence Research, Vol. 13, pages 227−303, 2000.

[2] T. G. Dietterich: *An Overview of MAXQ Hierarchical Reinforcement Learning.* Lecture Notes in Computer Science, Vol. 1864, pages 26−46, 2000.

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

[4] P. Spronck: *Adaptive Game AI.* PhD thesis, 2005.

[5] M. Buro and T. M. Furtak: *RTS games and real-time AI research.* Proceedings of the Behavior Representation in Modeling and Simulation Conference.

[6] C. J. C. H. Watkins and P. Dayan: *Q-learning.* Machine Learning, Vol. 8, pages 279−292, 1992.

[7] R. Parr and S. Russell: *Reinforcement Learning with Hierarchies of Machines.* Advances in Neural Information Processing Systems, Vol. 10, Cambridge, MA. MIT Press, 1998.

[8] R. S. Sutton, D. Precup, and S. Singh: *Between MDPs and Semi-MDPs: Learning, Planning, and Representing Knowledge at Multiple Temporal Scales.* Tech. rep., University of Massachusetts, Department of Computer and Information Sciences, Amherst, MA., 1998.

[9] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra: *Planning and Acting in Partially Observable Domains.* Artificial Intelligence Journal, Vol. 101, 1995.

[10] A.G. Barto and S. Mahadevan: *Recent Advances in Hierarchical Reinforcement Learning.* Discrete Event Dynamic Systems: Theory and Application, Vol. 13, pages 341−379, 2003.

[11] F. Huizinga: *Hierarchical Reinforcement Learning and Safe State Abstraction on Realtime Strategy Games.* Bachelor Thesis, 2007.