Concurrent Hierarchical Reinforcement Learning for RoboCup Keepaway

Aijun Bai†, Stuart Russell†, and Xiaoping Chen‡

† Computer Science Division, Univerisity of California at Berkeley
{aijunbai, russell}@berkeley.edu
‡ Department of Computer Science, University of Science and Technology of China
xpchen@ustc.edu.cn

Abstract. RoboCup Keepaway, originated from the RoboCup soccer simulation 2D challenge, has been widely used as a machine learning benchmark. In this paper, we present a concurrent hierarchical reinforcement learning approach to RoboCup Keepaway. Following the idea of *hierarchies of abstract machines* (HAMs), we write a partial policy as a HAM from the perspective of a single keeper, run multiple instances of the HAM, and use reinforcement learning to learn the optimal completion of the resulting joint HAM. Furthermore, we apply the idea of exploiting the intrinsic internal transitions within the HAM structure for more efficient learning. Experimental results confirm that the concurrent HAM approaches outperform the state of the art significantly on the very complex RoboCup Keepaway domain.

Keywords: Hierarchical Reinforcement Learning, HAM, RoboCup Keepaway

1 Introduction

Reinforcement learning (RL) tackles the problem of learning a rewarding behavior in an unknown environment via trial-and-error [18]. Recent advances in RL have led to great success on problems that pose significant challenges [12,15]. However, standard "flat" RL algorithms often learn slowly in environments requiring complex behaviors, due to the curses of dimensionality and history. *Hierarchical reinforcement learning* (HRL) aims to scale RL by incorporating prior knowledge about the structure of good policies into the algorithms [5]. Popular HRL solutions include the *options* theory [19], the *hierarchies of abstract machines* (HAMs) framework [14,1], and the *MAXQ* approach [7]. One of the major advantages of HRL approaches is the possibility of exploiting *temporal abstraction* and *hierarchical control*, where macro-actions following their own polices until termination.

In this paper, we focus on building intelligent agents that play the game of RoboCup Keepaway via hierarchical reinforcement learning. RoboCup Keepaway is a sub-task of the RoboCup soccer simulation 2D challenge [10,17]. It has been widely used as a machine learning benchmark [17], which presents significant challenges to machine learning methods, including continuous state and action spaces, multiple agents, and long and variable delays in the effects of actions. In RoboCup Keepaway, one team

of keepers merely seeks to keep control of the ball for as long as possible. Following the idea of HAMs, we write a partial policy as a HAM from the perspective of a single keeper, run multiple instances of the HAM, and use reinforcement learning to learn the optimal completion of the resulting joint HAM. We further apply the idea of HAMQ-INT [3], a novel HRL algorithm that identifies and exploits internal transitions within a HAM for efficient learning, to recursively shortcircuit the computation of Q values whenever applicable. We empirically confirm that HAMQ-INT outperforms the state of the art significantly on the benchmark RoboCup Keepaway domain. The main contribution of this paper is that we apply HAMQ-INT successfully to the RoboCup Keepaway domain, which, to the best of our knowledge, is the first application of the HAM framework to a very complex domain.

The remainder of the paper is organized as follows. Section 2 introduces some related work. Section 3 briefly reviews some background on RoboCup Keepaway and the HAM framework. Section 4 presents a concurrent HAM approach to RoboCup Keepaway. Section 5 presents the proposed HAMQ-INT algorithms. Section 6 describes the empirical result, and Section 7 concludes with discussion of future work.

2 Related Work

Stone *et al.* [16] develop a linear SARSA algorithm for RoboCup Keepaway following the options theory where the agent learns to select over a given set of low-level options. Along with some standard ball-controlling options, such as **Pass**() and **Hold**(), the keeper is also given a **GetOpen**() option which encodes the moving strategy when it is not controlling the ball. In their algorithm, each keeper learns separately assuming that other keepers and takers are part of the environment. In the experiment, we adopt their approach and develop an Option algorithm as one of the baselines. Kalyanakrishnan *et al.* [8] extend RoboCup Keepaway to Half Field Offense involving more agents and more complex behaviors. The authors notice that the Option algorithm has very sparse learning updates since each keeper is learning separately. They propose an inter-agent communication mechanism to facilitate information sharing among the agents and enable more frequent and reliable learning updates. In fact, since all the players begin with the same initial Q function and make the same updates, their action-value functions will always be alike, thereby reducing an essentially distributed problem to one of centralized control. In the experiment, we develop a concurrent-Option algorithm as an extension of this idea, where a global Q function is shared and maintained among all learners. Kalyanakrishnan *et al.* [9] extend the Option algorithm by including a specific learning component of the **GetOpen**() option. Their approach learns the option-selection policy over the ball-controlling options and the **GetOpen**() option iteratively: when one component is learning, the other one is kept unchanged. In contrast, our methods learn the two components simultaneously in a unified hierarchical reinforcement learning framework. Bai *et al.* [4] develop a MAXQ-based hierarchical planning algorithm for RoboCup domain. In this paper, we focus on hierarchical reinforcement learning instead.

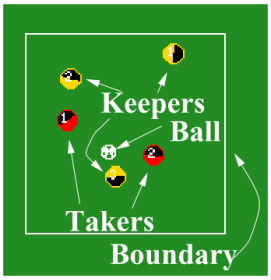


Fig. 1: A 3 vs. 2 instance of RoboCup Keepaway.

3 Background

In this section, we review some background on the RoboCup Keepaway task, reinforcement learning and the general HAM framework.

3.1 RoboCup Keepaway

In RoboCup Keepaway, a team of keepers tries to maintain the ball possession within a limited field, while a team of takers tries to take the ball. Figure 1 shows an instance of Keepaway with 3 keepers and 2 takers. The system has continuous state and action spaces. A state encodes positions and velocities for the ball and all players. At each time step (within 100 ms), a player can execute a parametrized primitive action, such as turn($angle$), dash($power$) or kick($power$, $angle$), where the turn action changes the body angle of the player, the dash action gives an acceleration to the player, and the kick action gives an acceleration to the ball if the ball is within the maximal kickable area of the player. All primitive actions are exposed to noises. Each episode begins with the ball and all players at fixed positions, and ends if any taker kicks the ball, or the ball is out of the field. The cumulative reward for the keepers is the total number of time steps for an episode. Instead of learning to select between primitive actions, the players are provided with a set of programmed options/skills including: 1) **Stay**() remaining stationary at the current position; 2) **Move**(d, v) dashing towards direction d with speed v; 3) **Intercept**() intercepting the ball; 4) **Pass**(k, v) passing the ball to teammate k with speed v; and 5) **Hold**() remaining stationary while keeping the ball kickable. The takers are assumed to follow fixed policies. The goal in RoboCup Keepaway is then to learn best-response policies for keepers on top of the provided low-level skills.

3.2 Reinforcement Learning with Hierarchies of Machines

Reinforcement Learning *Reinforcement learning* (RL) usually tackles the problem of learning a rewarding behavior in an unknown environment modeled as a *Markov decision process* (MDP). Formally, an MDP is a tuple $\langle S, A, T, R, \gamma \rangle$, where S and A are the state and action spaces, $T(s'|s, a)$ and $R(s, a)$ are the transition and reward functions, and γ is a discount factor [6]. The goal for an MDP is to find an *optimal*

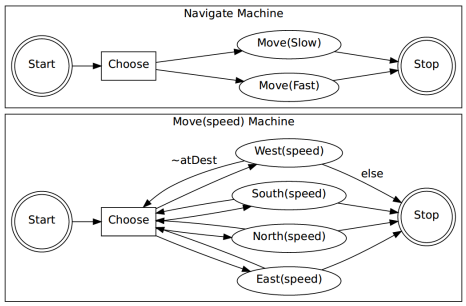


Fig. 2: An example of a HAM for a mobile robot.

policy $\pi^* : S \rightarrow A$ that maximizes the expected cumulative reward. In the setting of reinforcement learning, an agent learns an optimal policy by interacting with its environment. A Q learning agent achieves this by performing Q update, once it reaches state s' with reward r after executing action a in state s:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a')\right), \tag{1}$$

where α is a learning rate. *Semi Markov decision processes* (SMDPs) allow for actions that take multiple time steps to terminate. The transition function for an SMDP has the form $T(s', N|s, a)$, where N is the number of time steps that action a takes. Similarly, the Q update rule for a SMDP is:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma^{\tau} \max_{a'} Q(s',a')\right), \tag{2}$$

where τ is number of time steps elapsed after executing action a in state s and before reaching state s', and r is the cumulative reward in-between.

The HAM Approach The idea of HAM is to encode a partial policy for an agent as a set of hierarchical finite state machines with unspecified choice states, and use RL to learn its optimal completion. We adopt a different definition of HAM, allowing arbitrary call graph, despite the original definition of Parr and Russell [14] which requires that the call graph is a tree. Formally, a HAM $\mathcal{H} = \{\mathcal{N}_0, \mathcal{N}_1, \dots\}$ consists of a set of Moore machines $\mathcal{N}_i$ [13], where $\mathcal{N}_0$ is the root machine which serves as the starting point of the agent. A machine $\mathcal{N}$ is a tuple $\langle M, \Sigma, \Lambda, \delta, \mu\rangle$, where M is the set of machine states, Σ is the input alphabet which corresponds to the environment state space S, Λ is the output alphabet, δ is the machine transition function with $\delta(m, s)$ being the next machine state given machine state $m \in M$ and environment state $s \in S$, and μ is the machine output function with $\mu(m) \in \Lambda$ being the output of machine state $m \in M$. There are 5 types of machine states: **start** states are the entries of running machines; **action** states execute an action in the environment; **choose** states nondeterministically select the next machine states; **call** states invoke the execution of other machines; and, **stop** states end current machines and return control to calling machines. A machine

```
Run (𝒩 : machine, z : stack, s : environment state) :
z.Push (𝒩)
m ← 𝒩.start
while m ≠ 𝒩.stop do
    if Type (m) = action then
        s ← Execute (μ(m))
    else if Type (m) = call then
        s ← Run (μ(m), z, s, π)
    if Type (m) = choose then
        z.Push (m)
        m ← Choose (z, s, μ(m))
        z.Pop ()
    else
        m ← δ(m, s)
z.Pop ()
return s
```

Algorithm 1: Running a HAM.

$\mathcal{N}$ has uniquely one **start** state and one **stop** state, referred as $\mathcal{N}.start$ and $\mathcal{N}.stop$ respectively. For **start** and **stop** states, the outputs are not defined; for **action** states, the outputs are the associated primitive actions; for **call** states, the outputs are the next machines to run; and, for **choose** states, the outputs are the sets of possible choices, where each choice corresponds to a next machine state. For an example, see Figure 2, which shows a HAM for a mobile robot navigating in a grid map. The Navigate machine has a choice state, at which it has to choose between Move(Fast) and Move(Slow). The Move($speed$) machine has to select repeatedly between East, West, South and North with specified $speed$ parameter until the robot is at its destination.

To run a HAM $\mathcal{H}$, a run-time stack (or stack for short) is needed. Each frame of this stack stores run-time information such as the active machine, its machine state, the parameters passing to this machine and the values of local variables used by this machine. Algorithm 1 gives the pseudo-code for running a HAM, where the **`Execute`** function executes an action in the environment and returns the next environment state, and the **`Choose`** function picks the next machine state given the updated stack z, the current environment state s and the set of available choices $\mu(m)$. Let $\mathcal{Z}$ be the space of all possible stacks given HAM $\mathcal{H}$. It has been shown that an agent running a HAM $\mathcal{H}$ over an MDP $\mathcal{M}$ yields a joint SMDP $\mathcal{H} \circ \mathcal{M}$ defined over the joint space of S and $\mathcal{Z}$. The only actions of $\mathcal{H} \circ \mathcal{M}$ are the choices allowed at choice points. A choice point is a joint state (s, z) with z.**`Top`**`()` being a **choose** state. This is an SMDP because once a choice is made at a choice point, the system — the composition of $\mathcal{H}$ and $\mathcal{M}$ — runs automatically until the next choice point is reached. The policy of this SMDP implements exactly the **`Choose`** function in Algorithm 1. An optimal policy of this SMDP corresponds to an optimal completion of the input HAM, which can be found by applying a HAMQ algorithm [14]. HAMQ keeps track of the previous choice point (s, z), the choice made c and the cumulative reward r thereafter. Whenever it enters

```
Navigate(s : environment state):
speed ← Choose_1(Slow, Fast)
s ← Move(s, speed)
return s

Move(s : environment state, speed : parameter):
while not s.atDest() do
    a ← Choose_2(West, South, North, East)
    s ← Execute(a, speed)
return s
```

Algorithm 2: A HAM in pseudo-code for a mobile robot.

into a new choice point (s', z'), it performs the SMDP Q update as follows:

$$Q(s, z, c) \leftarrow (1 - \alpha)Q(s, z, c) + \alpha \left(r + \gamma^{\tau} \max_{c'} Q(s', z', c') \right),$$

where τ is the number of steps between the two choice points.

As suggested by the language of ALisp [2], a HAM can be equivalently converted into a piece of code in modern programming languages, with *call-and-return* semantics and built-in routines for explicitly updating stacks, executing actions and getting new environment states. The execution of a HAM can then be simulated by running the code itself. This conversion is important, as it provides a much more efficient way of designing and running a HAM. For example, the HAM shown in Figure 2 is equivalent to the pseudo-code in Algorithm 2, where a machine becomes a function. Here, **`Execute`** is the macro executing an action with specified parameters and returning the next environment state; the **`Choose`** macro extends the **`Choose`** function from Algorithm 1 to choose among not only a set of machine states, but also a set of parameters for the next machine. Bookkeeping codes for maintaining the stack are omitted for simplicity. Marthi *et al.* show that [11] multiple HAMs can be ran concurrently to form a joint HAM, and the same reinforcement learning technique (namely the HAMQ algorithm) can be applied to learn the optimal completion of the resulting joint HAM provided with appropriate synchronization semantics among the concurrently running HAMs.

4 The HAM Approach to RoboCup Keepaway

We develop the partial policy represented as a HAM from the perspective of a single keeper, and run multiple instances of this HAM concurrently for each keeper to form a joint policy for all keepers following the proposal of [11]. To run multiple HAMs concurrently, they have to be synchronized, such that if any machine is at its **choose** state, the other machines have to wait; if multiple machines are at their **choose** states, a joint choice is made instead of independent choice for each machine. For this purpose, players have to share their learned value functions and the selected joint choice. A joint Q update is developed to learn the joint choice selection policy as an optimal completion of the resulting joint HAM. Algorithm 3 shows the HAM written in pseudo-code for a single keeper. Here, **`Keeper`** is the root machine. The **`Run`** macro runs a

```
Keeper (s : environment state) :
while not s.Terminate () do
    if s.BallKickable () then
        m ← Choose_1 (Pass, Hold) ; s ← Run (m, s)
    else if s.FastestToBall () then
        s ← Intercept (s)
    else
        m ← Choose_2 (Stay, Move) ; s ← Run (m, s)
return s

Pass (s : environment state) :
k ← Choose_3 (1, 2, ...) ; v ← Choose_4 (Normal, Fast)
while s.BallKickable () do
    s ← Run (Pass, k, v)
return s

Hold (s : environment state) :
s ← Run (Hold)
return s

Intercept (s : environment state) :
s ← Run (Intercept)
return s

Stay (s : environment state) :
i ← s.TmControlBall ()
while i = s.TmControlBall () do
    s ←Run (Stay)
return s

Move (s : environment state) :
d ← Choose_5 (0°, 90°, 180°, 270°) ; v ← Choose_6 (Normal, Fast)
i ← s.TmControlBall ()
while i = s.TmControlBall () do
    v ←Run (Move, d, v)
return s
```

Algorithm 3: The HAM for RoboCup Keepaway.

machine or an option with specified parameters. **BallKickable**, **FastestToBall**, **TmControlBall** are predicates used to determine the transition inside a machine. It is worth noting that the **Move** machine only considers 4 directions, with direction 0° being the direction towards the ball, and so on.

5 Efficient Learning by Leveraging Internal Transitions

In this section, we introduce the concept of internal transition, and develop a hierarchical reinforcement learning algorithm that automatically identifies and exploits internal transitions for efficient learning, following the idea of Bai *et al.* [3].

5.1 Internal Transitions within HAMs

It has been observed that a HAM with deep hierarchical structure, where there are many calls from a parent machine to one of its child machines over the hierarchy, induces many internal transitions. An internal transition is a transition over the joint state space, where only the run-time stack changes but the environment state does not. Internal transitions always come with zero rewards and deterministic outcomes in the resulting SMDP. Take the HAM for RoboCup Keepaway in Algorithm 3 as an example, there are many internal transitions within this single HAM. For example, when the **`Pass`** machine is selected at the choice point **`Choose`**$_1$ of the **`Keeper`** machine, the next 2 consecutive choice points must be **`Choose`**$_3$ and **`Choose`**$_4$ within the the **`Pass`** machine. When multiple HAMs are executing concurrently according to the concurrent schema shown in [11], there are even more internal transitions in the resulting joint HAM. For example, in a scenario of the 3 vs. 2 Keepaway game, where only keeper 1 can kick the ball, suppose the joint machine state is [**`Choose`**$_1$, **`Choose`**$_2$, **`Choose`**$_2$] with each element being the machine state of a HAM. If the joint choice made is [**`Pass`**, **`Move`**, **`Stay`**], then the next 2 consecutive machine states must be [**`Choose`**$_3$, **`Choose`**$_5$, **`Stay`**] and [**`Choose`**$_4$, **`Choose`**$_6$, **`Stay`**].

In general, the transition function of the resulting SMDP induced by running a HAM has the form $T(s', z', \tau | s, z, c) \in [0, 1]$, where (s, z) is the current choice point, c is the choice made, (s', z') is the next choice point, and τ is the number of time steps. Given a HAM with a deep hierarchy of machines, it is usually the case that there is no real actions executed between two consecutive choice points, therefore the number of time steps and the cumulative reward in-between are essentially zero. We call this kind of transition an internal transition, because the machine state changes but the environment state does not. Formally, a transition is a tuple $\langle s, z, c, r, s', z' \rangle$ with r being the cumulative reward. For an internal transition, we must have $s' = s$ and $r = 0$. In addition, because the dynamics of the HAM after a choice has been made and before an action is executed is deterministic by design, the next choice point (s, z') of an internal transition is deterministically conditioned only on $\langle s, z, c \rangle$. Let $\rho(s, z, c)$ be the $\mathcal{Z}$ component of the next choice point. If $\langle s, z, c \rangle$ leads to an internal transition, we must have $T(s, \rho(s, z, c), 0 | s, z, c) = 1$. Therefore, we have

$$\begin{aligned} Q(s, z, c) &= V(s, \rho(s, z, c)) \\ &= \max_{c'} Q(s, \rho(s, z, c), c'). \end{aligned} \tag{3}$$

So, we can store the rules of internal transition as $\langle s, z, c, z' \rangle$ tuples, where $z' = \rho(s, z, c)$. They can be used to recursively compute Q values according to Equation 3 when applicable. The size of the set of stored rules can be further reduced, because the machine transition function δ of a HAM is usually determined by a set of predicates defined over environment state s, rather than the exact values of all state variables. For example, the machine transition function of machine Move($speed$) in Figure 2 depends only on the value of atDest(s) for any state s. Suppose $\langle s_1, z, c \rangle$ leads to an internal transition with (s_1, z') being the next choice point. Let the set of predicates used to determine the trajectory in terms of active machines and machine states from z.**`Top`**`()` to z'.**`Top`**`()` be $\mathcal{P} = \{P_1, P_2, \dots\}$. Let the value of $\mathcal{P}$ given state s be

```
QUpdate(s' : state, z' : stack, r : reward, t' : current time, P : evaluated predicates) :
if t' = t then
    ρ[P, P(s), z, c] ← z'
else
    QTable(s, z, c) ← (1 − α) QTable(s, z, c) + α(r + γ^{t'−t} max_{c'} Q(s', z', c'))
(t, s, z) ← (t', s', z')

Q(s : state, z : stack, c : choice) :
if ∃P s.t. ⟨P, P(s), z, c⟩ ∈ ρ.Keys() then
    q ← −∞
    z' ← ρ[P, P(s), z, c]
    for c' ∈ μ(z.Top()) do
        q ← max(q, Q(s, z', c'))
    return q
else
    return QTable(s, z, c)
```

Algorithm 4: The HAMQ-INT algorithm.

$\mathcal{P}(s) = \{P_1(s), P_2(s), \dots\}$. It can be concluded that the transition trajectory induced by $\mathcal{P}$ depends only on $\mathcal{P}(s_1)$, after choice c is made at choice point (s_1, z). On the other hand, if the set of predicates $\mathcal{P}$ over state s_2 ($s_2 \neq s_1$) has the same value as of state s_1, namely $\mathcal{P}(s_2) = \mathcal{P}(s_1)$, and the same choice c is made at choice point (s_2, z), then the followed transition trajectory before reaching the next choice point must also be the same as of $\langle s_1, z, c\rangle$. In other words, $\langle s_2, z, c\rangle$ leads to an internal transition such that $\rho(s_1, z, c) = \rho(s_2, z, c)$.

Thus, the rule of internal transition $\langle s_1, z, c, z'\rangle$ can be equivalently stored and retrieved as $\langle \mathcal{P}, \mathcal{P}(s_1), z, c, z'\rangle$, which automatically applies to $\langle s_2, z, c, z'\rangle$, if $\mathcal{P}(s_2) = \mathcal{P}(s_1)$. Here, z' is the stack of the next choice point such that $z' = \rho(s_1, z, c) = \rho(s_2, z, c)$. The size of the joint space of encountered predicates and their values is determined by the HAM itself, which is typically much smaller than the size of the state space. For example, for a problem with continuous state space (such as the RoboCup Keepaway domain we considered), this joint space is still limited. In summary, we can have an efficient way of storing and retrieving the rules of internal transition by keeping track of the predicates evaluated between two choice points.

5.2 The HAMQ-INT Algorithm

The main idea of HAMQ-INT is to identify and take advantage of internal transitions within a HAM. For this purpose, HAMQ-INT automatically keeps track of the predicates that are evaluated between two choice points, stores the discovered rules of internal transition based on predicates and the corresponding values, and uses the learned rules to shortcircuit the computation of Q values whenever it is possible. To detect internal transitions, a global environment time t is maintained. It is incremented by one only when there is an action executed in the environment. When the agent enters a choice point (s', z') after having made a choice c at choice point (s, z), and finds that t is not

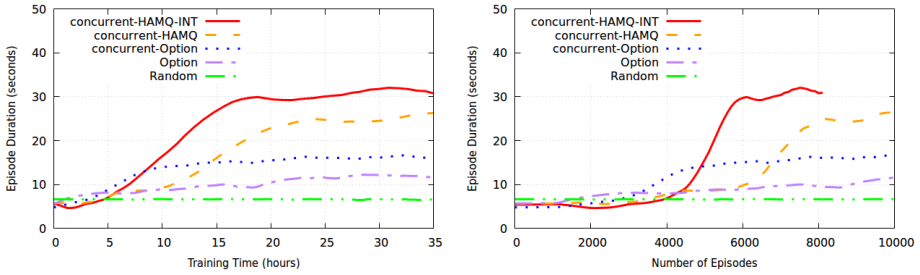
incremented since the previous choice point, it must be the case that $s' = s$ and $\langle s, z, c\rangle$ leads to an internal transition. Let $\mathcal{P}$ be the set of predicates that have been evaluated between these two choice points. Then a new rule of internal transition $\langle \mathcal{P}, \mathcal{P}(s), z, c, z'\rangle$ is found. The agent can conclude that for any state x, if $\mathcal{P}(x) = \mathcal{P}(s)$, then $\langle x, z, c\rangle$ leads to an internal transition as well. In the implementation, the agent uses a hash table ρ to store the learned rules, such that $\rho[\mathcal{P}, \mathcal{P}(s), z, c] = z'$, if $\langle \mathcal{P}, \mathcal{P}(s), z, c, z'\rangle$ is a rule of internal transition. One thing to note is that, because z' is deterministically conditioned on $\langle \mathcal{P}, \mathcal{P}(s), z, c\rangle$ for an internal transition, the value of $\rho[\mathcal{P}, \mathcal{P}(s), z, c]$ will not be changed after it has been updated for the first time.

When the agent needs to evaluate a Q function, say $Q(s, z, c)$, and finds that $\langle s, z, c\rangle$ leads to an internal transition according to the current learned rules, Equation 3 is used to decompose $Q(s, z, c)$ into the Q values of the next choice points, which are evaluated recursively in the same way, essentially leading to a tree of exact Bellman backups. In fact, only the terminal Q values of this tree needs to be learned, enabling efficient learning for the agent. Algorithm 4 gives the pseudo-code of the HAMQ-INT algorithm. Here, the **`QTable`** function returns the stored Q value as request. It can be implemented in either tabular or function approximation ways. The **`Q`** function evaluates the Q value of (s, z, c) tuple. It first checks whether (s, z, c) subjects to any learned internal transition rule. This is done by checking whether there exists an encountered set of predicates $\mathcal{P}$, such that $\langle \mathcal{P}, \mathcal{P}(s), z, c\rangle \in \rho$.**`Keys`**`()`. The uniqueness of transition trajectory for an internal transition ensures that there will be at most one such $\mathcal{P}$. If there is such $\mathcal{P}$, **`Q`** uses the retrieved rule to recursively decompose the requested Q value according to Equation 3; otherwise, it simply returns the stored Q value by querying **`QTable`**.

The **`QUpdate`** function performs the SMDP Q update. It is called once the agent enters a new choice point. The caller has to keep track of the current state s', the current stack z', the evaluated predicates $\mathcal{P}$ on state since the previous choice point and the cumulative reward r in-between. If the current time t' equals to the time t of the previous choice point, it must be the case that $\langle s, z, c, 0, s, z'\rangle$ is an internal transition. Thus, a new rule $\langle \mathcal{P}, \mathcal{P}(s), z, c\rangle$ is learned, and the ρ table is updated accordingly. If $t' \neq t$, meaning there are some actions executed in the environment, it simply performs the Q update. Finally, it uses the current (t', s', z') tuple to update the (global) previous (t, s, z) tuple, so the function will be prepared for the next call.

6 Experiment

We compare concurrent-HAMQ-INT, concurrent-HAMQ, concurrent-Option, Option and Random algorithms. The Option algorithm is adopted from [16], where the agent learns an option-selection policy over **Hold**() and **Pass**(k, v) options if it can kick the ball, otherwise it follows a fixed policy: if it is the fastest one to intercept the ball, it intercepts; otherwise, it follows a **GetOpen**() option. The **GetOpen**() option, which enables the agent to move to an open area in the field, is manually programmed beforehand. In the original Option learning algorithm, each agent learns independently. We argue that this setting is problematic, since it actually incorrectly assumes that other keepers are stationary. We extend Option to concurrent-Option, by sharing the learned value functions and the option selected. The HAM algorithms are not provided with



(a) (b)

Fig. 3: Experimental result on a 3 vs. 2 instance of RoboCup Keepaway evaluated over (a) training time and (b) number of episodes. Two short videos showing the initial and converged policies of HAMQ-INT can be found at links 1 and 2 respectively.

the **GetOpen**() option. Instead, they have to learn their own versions of **GetOpen**() by selecting from **`Stay`** and **`Move`** machines. Each taker follows the same fixed policy: it always tries to intercept the ball. The Random algorithm is a non-learning version of Option, which selects available options randomly.

The SARSA(λ)-learning rule with a linear function approximator is used to implement the SMDP Q update for all learning algorithms. A state is represented as a vector of 15 features consisting of some distances and angles calculated from the state, which is then encoded as a huge binary vector following the *tile coding* technique. The resulting binary vector has approximately 50,000 bits but is constrained to have only with 480 ones. For HAMQ approaches, a dynamically calculated hash value of the run-time stack is used as its index. The learning rate α and the eligibility decaying rate λ are set to be 0.125 and 0.5 respectively. Figure 3a and 3b show the experiment result on a 3 vs. 2 instance of RoboCup Keepaway evaluated over training time and number of episodes respectively. The data points are averaged using a moving window with size of 1000 episodes. The training time is accounted from the perspective of the simulated world where a step/cycle takes exactly 100 ms. Provided with the same training time, algorithms with better learning performance will result to less episodes, since they learn to reliably control the ball very soon. It can be seen from the result that concurrent-Option outperforms Option significantly, concurrent-HAMQ outperforms concurrent-Option after about 15 hours of training, and concurrent-HAMQ-INT has the best performance. We expect the reason to be 1) significantly less number of Q values have to be learned because many of them are recursively decomposed into the combinations of other Q values following the Bellman backup tree induced by recursively applying the learned internal transition rules; and 2) one learned abstract internal transition rule from one transition can be applied to unlimited number of other states in the case of continuous state space.

7 Conclusions

In this paper, we present a concurrent hierarchical reinforcement learning approach to the benchmark RoboCup Keepaway domain. We apply the idea of HAMQ-INT that automatically discovers and exploits internal transitions within a HAM for efficient learning. We empirically confirm that the concurrent HAM approaches outperform the state of the art significantly on RoboCup Keepaway. In future work, we would like to apply this idea to the full RoboCup soccer simulation game and other challenging domains that require very complex behaviors. The way we taking advantage of internal transitions within a HAM can be seen as leveraging some prior knowledge of the transition model of a reinforcement learning problem, which happens to be deterministic. We would also like to extend this internal transition idea to more general reinforcement learning problems, where models are partially known in advance.

8 Acknowledgments

Funding for this research was provided by ONR under contract N00014-12-1-0609, and by DARPA under contract N66001-15-2-4048. Opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the funding agencies. The authors would like to thank the WrightEagle soccer simulation team (particularly, Changjie Fan, Feng Wu, Ke Shi, Haochong Zhang and Guanghui Lu) for contributing to the base code used in the experiment. The authors would also thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Andre, D., Russell, S.J.: Programmable reinforcement learning agents. Advances in neural information processing systems pp. 1019–1025 (2001)
2. Andre, D., Russell, S.J.: State abstraction for programmable reinforcement learning agents. In: Proceedings of the 8th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence. pp. 119–125 (2002)
3. Bai, A., Russell, S.J.: Efficient reinforcement learning with hierarchies of machines by leveraging internal transitions. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19 - 25 (2017)
4. Bai, A., Wu, F., Chen, X.: Online planning for large Markov decision processes with hierarchical decomposition. ACM Transactions on Intelligent Systems and Technology 6(4), 45 (2015)
5. Barto, A., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems 13, 341–379 (2003)
6. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton, NJ, USA (1957)
7. Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of Machine Learning Research 13(1), 63 (May 1999)
8. Kalyanakrishnan, S., Liu, Y., Stone, P.: Half field offense in robocup soccer: A multiagent reinforcement learning case study. RoboCup 2006: Robot Soccer World Cup X pp. 72–85 (2007)

9. Kalyanakrishnan, S., Stone, P.: Learning complementary multiagent behaviors: A case study. In: Robot Soccer World Cup. pp. 153–165. Springer (2009)
10. Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., Asada, M.: The robocup synthetic agent challenge 97. In: RoboCup-97: Robot Soccer World Cup I, pp. 62–73. Springer (1998)
11. Marthi, B., Russell, S.J., Latham, D., Guestrin, C.: Concurrent hierarchical reinforcement learning. In: IJCAI. pp. 779–785 (2005)
12. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (2015)
13. Moore, E.F.: Gedanken-experiments on sequential machines. Automata studies 34, 129–153 (1956)
14. Parr, R., Russell, S.: Reinforcement learning with hierarchies of machines. In: Advances in Neural Information Processing Systems. vol. 10 (1998)
15. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. Nature 529(7587), 484–489 (2016)
16. Stone, P., Sutton, R., Kuhlmann, G.: Reinforcement learning for robocup soccer keepaway. Adaptive Behavior 13(3), 165–188 (2005)
17. Stone, P., Kuhlmann, G., Taylor, M.E., Liu, Y.: Keepaway soccer: From machine learning testbed to benchmark. In: Robot Soccer World Cup. pp. 93–105. Springer (2005)
18. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, vol. 1. MIT press Cambridge (1998)
19. Sutton, R., Precup, D., Singh, S.: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence 112(1), 181–211 (1999)