Towards a Principled Solution to Simulated Robot Soccer

Aijun Bai, Feng Wu, and Xiaoping Chen

Department of Computer Science,
University of Science and Technology of China,
{baj, wufeng}@mail.ustc.edu.cn, xpchen@ustc.edu.cn

Abstract. The RoboCup soccer simulation 2D domain is a very large testbed for the research of planning and machine learning. It has competed in the annual world championship tournaments in the past 15 years. However it is still unclear that whether more principled techniques such as decision-theoretic planning take an important role in the success for a RoboCup 2D team. In this paper, we present a novel approach based on MAXQ-OP to automated planning in the RoboCup 2D domain. It combines the benefits of a general hierarchical structure based on MAXQ value function decomposition with the power of heuristic and approximate techniques. The proposed framework provides a principled solution to programming autonomous agents in large stochastic domains. The MAXQ-OP framework has been implemented in our RoboCup 2D team, WrightEagle. The empirical results indicated that the agents developed with this framework and related techniques reached outstanding performances, showing its potential of scalability to very large domains.

Keywords: RoboCup, Soccer Simulation 2D, MAXQ-OP

1 Introduction

As one of oldest leagues in RoboCup, soccer simulation 2D has achieved great successes and inspired many researchers all over the world to engage themselves in this game each year [5]. Hundreds of research articles based on RoboCup 2D have been published.[1] Comparing to other leagues in RoboCup, the key feature of RoboCup 2D is the abstraction made, which relieves the researchers from having to handle low-level robot problems such as object recognition, communications, and hardware issues. The abstraction enables researchers to focus on high-level functions such as cooperation and learning. The key challenge of RoboCup 2D lies in the fact that it is a fully distributed, multi-agent stochastic domain with continuous state, action and observation space [8].

Stone *et al.* [9] have done a lot of work on applying reinforcement learning methods to RoboCup 2D. Their approaches learn high-level decisions in a keepaway subtask using episodic SMDP Sarsa(λ) with linear tile-coding function approximation. More precisely, their robots learn individually when to hold the ball and when to pass it to a teammate. Most recently, they extended their work

[1] http://www.cs.utexas.edu/~pstone/tmp/sim-league-research.pdf

to a more general task named half field offense [6]. On the same reinforcement learning track, Riedmiller *et al.* [7] have developed several effective techniques for learning mainly low-level skills in RoboCup 2D.

In this paper, we present an alternative approach based on MAXQ-OP [1] to automated planning in the RoboCup 2D domain. It combines the main advantages of online planning and hierarchical decomposition, namely MAXQ. The proposed framework provides a principled solution to programming autonomous agents in large stochastic domains. The key contribution of this paper lies in the overall framework for exploiting the hierarchical structure online and the approximation made for computing the *completion function*. The MAXQ-OP framework has been implemented in our team WrightEagle, which has been participating in annual competitions of RoboCup since 1999 and have got 3 champions and 4 runners-up of RoboCup in recent 7 years.[2] The empirical results indicated that the agents developed with this framework and the related techniques reached outstanding performances, showing its potential of scalability to very large domains.

The remainder of this paper is organized as follows. Section 2 introduces some background knowledge. Section 3 describes the MAXQ-OP framework in detail. Section 4 presents the implementation details in the RoboCup 2D domain, and Section 5 shows the empirical evaluation results. Finally, Section 6 concludes the paper with some discussion of future work.

2 Background

In this section, we briefly introduce the background, namely RoboCup 2D and the MAXQ hierarchical decomposition methods. We assume that readers already have sufficient knowledge on RoboCup 2D. For MAXQ, we only describe some basic concepts but refer [4] for more details.

2.1 RoboCup soccer simulation 2D

In RoboCup 2D, a central *server* simulates a 2-dimensional virtual soccer field in real-time. Two teams of fully autonomous agents connect to the server via network sockets to play a soccer game over 6000 steps. A team can have up to 12 clients including 11 players (10 fielders plus 1 goalie) and a coach. Each client interacts independently with the server by 1) receiving a set of observations; 2) making a decision; and 3) sending actions back to the server. Observations for each player only contain noisy and local geometric information such as the distance and angle to other players, ball, and field markings within its view range. Actions are atomic commands such as turning the body or neck to an angle, dashing in a given direction with certain power, kicking the ball to an angle with specified power, or slide tackling the ball.

2.2 MAXQ hierarchical decomposition

Markov decision processes (MDPs) have been proved to be a useful model for planning under uncertainty. In this paper, we concentrate on undiscounted goal-

[2] Team website: `http://www.wrighteagle.org/2d`

directed MDPs (also known as *stochastic shortest path problems*). It is shown that any MDP can be transformed into an equivalent undiscounted negative goal-directed MDP where the reward for non-goal states is strictly negative [2]. So undiscounted goal-directed MDP is actually a general formulation.

The MAXQ technique decomposes a given MDP M into a set of sub-MDPs arranged over a hierarchical structure, denoted by $\{M_0, M_1, \cdots, M_n\}$. Each sub-MDP is treated as a distinct subtask. Specifically, M_0 is the root subtask which means solving M_0 solves the original MDP M. An *unparameterized* subtask M_i is defined as a tuple $\langle T_i, A_i, \tilde{R}_i \rangle$, where:

- T_i is the *termination predicate* that defines a set of active states S_i, and a set of terminal states G_i for subtask M_i.
- A_i is a set of actions that can be performed to achieve subtask M_i, which can either be primitive actions from M, or refer to other subtasks.
- $\tilde{R}_i$ is the optional *pseudo-reward function* which specifies pseudo-rewards for transitions from active states S_i to terminal states G_i.

It is worth pointing out that if a subtask has task parameters, then different binding of the parameters, may specify different instances of a subtask. Primitive actions are treated as primitive subtasks such that they are always executable, and will terminate immediately after execution.

Given the hierarchical structure, a *hierarchical policy* π is defined as a set of policies for each subtask $\pi = \{\pi_0, \pi_1, \cdots, \pi_n\}$, where π_i is a mapping from active states to actions $\pi_i : S_i \rightarrow A_i$. The *projected value function* of policy π for subtask M_i in state s, $V^\pi(i, s)$, is defined as the expected value after following policy π at state s until the subtask M_i terminates at one of its terminal states in G_i. Similarly, $Q^\pi(i, s, a)$ is the expected value by firstly performing action M_a at state s, and then following policy π until the termination of M_i. It is worth noting that $V^\pi(a, s) = R(s, a)$ if M_a is a primitive action $a \in A$.

Dietterich [4] has shown that a *recursively optimal policy* π^* can be found by recursively computing the optimal projected value function as:

$$Q^*(i, s, a) = V^*(a, s) + C^*(i, s, a), \tag{1}$$

where

$$V^*(i, s) = \begin{cases} R(s, i) & \text{if } M_i \text{ is primitive} \\ \max_{a \in A_i} Q^*(i, s, a) & \text{otherwise} \end{cases}, \tag{2}$$

and $C^*(i, s, a)$ is the *completion function* fot optimal policy π^* that estimates the cumulative reward received with the execution of (macro-) action M_a before completing the subtask M_i, as defined below:

$$C^*(i, s, a) = \sum_{s', N} \gamma^N P(s', N|s, a) V^*(i, s'), \tag{3}$$

where $P(s', N|s, a)$ is the probability that subtask M_a at s terminates at state s' after N steps.

3 Online Planning with MAXQ

In this section, we explain in detail how our MAXQ-OP solution works. As mentioned above, MAXQ-OP is a novel online planning approach that incorporates the power of the MAXQ decomposition to efficiently solve large MDPs.

Algorithm 1: `OnlinePlanning()`

Input: an MDP model with its MAXQ hierarchical structure
Output: the accumulated reward r after reaching a goal

```
r ← 0;
s ← GetInitState();
while s ∉ G_0 do
    ⟨v, a_p⟩ ← EvaluateState(0, s, [0, 0, ··· , 0]);
    r ← r+ ExecuteAction(a_p, s);
    s ← GetNextState();
return r;
```

3.1 Overview of MAXQ-OP

In general, online planning interleaves planning with execution and chooses the best action for the current step. Given the MAXQ hierarchy of an MDP, $M = \{M_0, M_1, \cdots, M_n\}$, the main procedure of MAXQ-OP evaluates each subtask by forward search to compute the recursive value functions $V^*(i, s)$ and $Q^*(i, s, a)$ online. This involves a complete search of all paths through the MAXQ hierarchy starting from the root task M_0 and ending with some primitive subtasks at the leaf nodes. After the search process, the best action $a \in A_0$ is chosen for the root task based on the recursive Q function. Meanwhile, the best primitive action $a_p \in A$ that should be performed first is also determined. This action a_p will be executed to the environment, leading to a transition of the system state. Then, the planning procedure starts over to select the best action for the next step.

As shown in Algorithm 1, state s is initialized by `GetInitState` and the function `GetNextState` returns the next state of the environment after `ExecuteAction` is performed. It executes a primitive action to the environment and returns a reward for running that action. The main process loops over until a goal state in G_0 is reached. Obviously, the key procedure of MAXQ-OP is `EvaluateState`, which evaluates each subtask by depth-first search and returns the best action for the current state. Section 3.2 will explain `EvaluateState` in more detail.

3.2 Task Evaluation over Hierarchy

In order to choose the best action, the agent must compute a Q function for each possible action at the current state s. Typically, this will form a search tree starting from s and ending with the goal states. The search tree is also known as an AND-OR tree where the AND nodes are actions and the OR nodes are states. The root node of such an AND-OR tree represents the current state. The search in the tree is processed in a depth-first fashion until a goal state or a certain pre-determined fixed depth is reached. When it reaches the depth, a heuristic is often used to evaluate the long term value of the state at the leaf node.

When the task hierarchy is given, it is more difficult to perform such a search procedure since each subtask may contain other subtasks or several primitive actions. As shown in Algorithm 2, the search starts with the root task M_i and the current state s. Then, the node of the current state s is expanded by trying each possible subtask of M_i. This involves a recursive evaluation of the subtasks and the subtask with the highest value is selected. As mentioned in Section 2,

Algorithm 2: `EvaluateState`(i, s, d)

Input: subtask M_i, state s and depth array d
Output: $\langle V^*(i,s), \text{a primitive action } a_p^* \rangle$

```
if M_i is primitive then return ⟨R(s, M_i), M_i⟩;
else if s ∉ S_i and s ∉ G_i then return ⟨−∞, nil⟩;
else if s ∈ G_i then return ⟨0, nil⟩;
else if d[i] ≥ D[i] then return ⟨HeuristicValue(i, s), nil⟩;
else
    ⟨v*, a_p*⟩ ← ⟨−∞, nil⟩;
    for M_k ∈ Subtasks(M_i) do
        if M_k is primitive or s ∉ G_k then
            ⟨v, a_p⟩ ← EvaluateState(k, s, d);
            v ← v+ EvaluateCompletion(i, s, k, d);
            if v > v* then
                ⟨v*, a_p*⟩ ← ⟨v, a_p⟩;
    return ⟨v*, a_p*⟩;
```

the evaluation of a subtask requires the computation of the value function for its children and the completion function. The value function can be computed recursively. Therefore, the key challenge is to calculate the completion function.

Intuitively, the completion function represents the optimal value of fulfilling the task M_i after executing a subtask M_a first. According to Equation 3, the completion function of an optimal policy π^* can be written as:

$$C^*(i,s,a) = \sum_{s',N} \gamma^N P(s', N|s, a) V^{\pi^*}(i, s'), \tag{4}$$

where

$$P(s', N|s, a) = \sum_{\langle s, s_1, \ldots, s_{N-1} \rangle} P(s_1|s, \pi_a^*(s)) \cdot P(s_2|s_1, \pi_a^*(s_1)) \cdots P(s'|s_{N-1}, \pi_a^*(s_{N-1})). \tag{5}$$

More precisely, $\langle s, s_1, \ldots, s_{N-1} \rangle$ is a path from the state s to the terminal state s' by following the optimal policy $\pi_a^* \in \pi^*$. It is worth noticing that π^* is a recursive policy constructed by other subtasks. Obviously, computing the optimal policy π^* is equivalent to solving the entire problem. In principle, we can exhaustively expand the search tree and enumerate all possible state-action sequences starting with s, a and ending with s' to identify the optimal path. Obviously, this may be inapplicable for large domains. In Section 3.3, we will present a more efficient way to approximate the completion function.

Algorithm 2 summarizes the major procedures of evaluating a subtask. Clearly, the recursion will end when: 1) the subtask is a primitive action; 2) the state is a goal state or a state outside the scope of this subtask; or 3) a certain depth is reached, i.e. $d[i] \geq D[i]$ where $d[i]$ is the current forward search depth and $D[i]$ is the maximal depth allowed for subtask M_i. It is worth pointing out, different maximal depths are allowed for each subtask. Higher level subtasks may have smaller maximal depth in practice. If the subtask is a primitive action, the immediate reward will be returned as well as the action itself. If the search

Algorithm 3: `EvaluateCompletion`(i, s, a, d)

Input: subtask M_i, state s, action M_a and depth array d
Output: estimated $C^*(i, s, a)$

$\tilde{G}_a \leftarrow$ `ImportanceSampling`$(G_a,\ D_a)$;
$v \leftarrow 0$;
for $s' \in \tilde{G}_a$ **do**
    $d' \leftarrow d$;
    $d'[i] \leftarrow d'[i] + 1$;
    $v \leftarrow v + \frac{1}{|\tilde{G}_a|}$ `EvaluateState`(i, s', d');
return v;

reaches a certain depth, it also returns with a heuristic value for the long-term reward. In this case, a *nil* action is also returned, but it will never be chosen by higher level subtasks. If none of the above conditions holds, it will loop over and evaluate all the children of this subtask recursively.

3.3 Completion Function Approximation

To exactly compute the optimal completion function, the agent must know the optimal policy π^* first which is equivalent to solving the entire problem. However, it is intractable to find the optimal policy online due to the time constraint. When applying MAXQ-OP to large problems, approximation should be made to compute the completion function for each subtask. One possible solution is to calculate an approximate policy offline and then to use it for the online computation of the completion function. However, it may be also challenging to find a good approximation of the optimal policy if the domain is very large.

Notice that the term γ^N in Equation 3 is equal to 1 when $\gamma = 1$, which is the default setting of this paper. Given an optimal policy, the subtask will terminate at a certain goal state with the probability of 1 after several steps. To compute the completion function, the only term need to be considered is $P(s', N|s, a)$–a distribution over the terminal states. Given a subtask, it is often possible to directly approximate the distribution disregarding the detail of execution.

Based on these observations, we assume that each subtask M_i will terminate at its terminal states in G_i with a prior distribution of D_i. In principle, D_i can be any probability distribution associated with each subtask. Denoted by $\tilde{G}_a$ a set of sampled states drawn from prior distribution D_a using *importance sampling* [10] techniques, the completion function $C^*(i, s, a)$ can be approximated as:

$$C^*(i, s, a) \approx \frac{1}{|\tilde{G}_a|} \sum_{s' \in \tilde{G}_a} V^*(i, s'). \tag{6}$$

A recursive procedure is proposed to estimate the completion function, as shown in Algorithm 3. In practice, the prior distribution D_a–a key distribution when computing the completion function, can be improved by considering the domain knowledge. Take the robot soccer domain for example. The agent at state s may locate in a certain position of the field. Suppose s' is the goal state of successfully scoring the ball. Then, the agent may have higher probability to reach s' if it

directly dribbles the ball to the goal or passes the ball to some teammates who is near the goal, which is specified by the action a in the model.

3.4 Heuristic Search in Action Space

For some domains with large action space, it may be very time-consuming to enumerate all possible actions exhaustively. Hence it is necessary to introduce some heuristic techniques (including prune strategies) to speed up the search process. Intuitively, there is no need to evaluate those actions that are not likely to be better. In MAXQ-OP, this is done by implementing a iterative version of `Subtasks` function which dynamically selects the most promising action to be evaluated next with the tradeoff between exploitation and exploration. Different heuristic techniques can be used for different subtasks, such as A*, hill-climbing, gradient ascent, etc. The discussion of the heuristic techniques is beyond the scope of this paper, and the space lacks for a detailed description of it.

4 Implementation in RoboCup 2D

It is our long-term effort to apply the MAXQ-OP framework to the RoboCup 2D domain. In this section, we present the implementation details of the MAXQ-OP framework in WrightEagle.

4.1 RoboCup 2D as an MDP

In this section, we present the technical details on modeling the RoboCup 2D domain as an MDP. As mentioned, it is a partially-observable multi-agent domain with continuous state and action space. To model it as a fully-observable single-agent MDP, we specify the state and action spaces and the transition and reward functions as follows:

State Space We treat teammates and opponents as part of the environment and try to estimate the current state with sequences of observations. Then, the state of the 2D domain can be represented as a fixed-length vector, containing state variables that totally cover 23 distinct objects (10 teammates, 11 opponents, the ball, and the agent itself).

Action Space All primitive actions, like dash, kick, tackle, turn and turn_neck, are originally defined by the 2D domain. They all have continuous parameters, resulting a continuous action space.

Transition Function Considering the fact that autonomous teammates and opponents make the environment unpredictable, the transition function is not obvious to represent. In our team, the agent assumes that all other players share a same predefined behavior model: they will execute a random kick if the ball is kickable for them, or a random walk otherwise. For primitive actions, the underlying transition model for each atomic command is fully determined by the server as a set of *generative* models.

Reward Function The underlying reward function has a *sparse* property: the agent usually earns zero rewards for thousands of steps before ball scored or conceded, may causing that the forward search process often terminate without

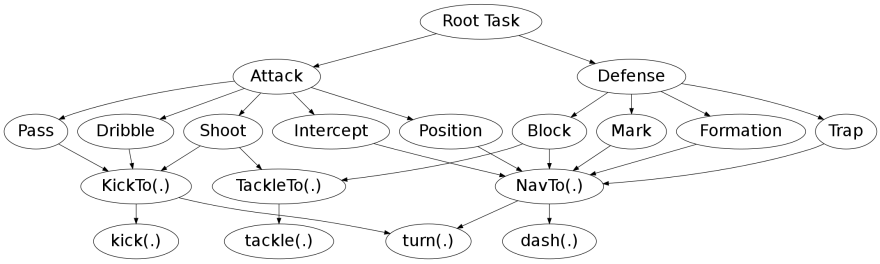

Fig. 1. MAXQ task graph for WrightEagle

any rewards obtained, and thus can not tell the differences between subtasks. In our team, to emphasize each subtask's characteristic and to guarantee that positive results can be found by the search process, a set of pseudo-reward functions is developed for each subtask.

To estimate the size of the state space, we ignore some secondary variables for simplification (such as heterogeneous parameters and stamina information). Totally 4 variables are needed to represent the ball's state including position (x, y) and velocity (v_x, v_y). In addition with (x, y) and (v_x, v_y), two more variables are used to represent each player's state including body direction d_b, and neck direction d_n. Therefore the full state vector has a dimensionality of 136. All these state variables have continuous values, resulting a high-dimensional continuous state space. If we discretize each state variable into 10^3 uniformly distributed values in its own field of definitions, then we obtain a simplified state space with 10^{408} states, which is extremely larger than domains usually studied in the literature.

4.2 Solution with MAXQ-OP

In this section, we describe how to apply MAXQ-OP to the RoboCup soccer simulation domain. Firstly, a series of subtasks at different levels are defined as the building blocks of constructing the MAXQ hierarchy, listed as follows:

- kick, turn, dash, and tackle: They are low-level parameterized primitive actions originally defined by the soccer server. A reward of -1 is assigned to each primitive action to guarantee that the optimal policy will try to reach a goal as fast as possible.
- KickTo, TackleTo, and NavTo: In the KickTo and TackleTo subtask, the goal is to kick or tackle the ball to a given direction with a specified velocity, while the goal of the NavTo subtask is to move the agent from its current location to a target location.
- Shoot, Dribble, Pass, Position, Intercept, Block, Trap, Mark, and Formation: These subtasks are high-level behaviors in our team where: 1) Shoot is to kick out the ball to score; 2) Dribble is to dribble the ball in an appropriate direction; 3) Pass is to pass the ball to a proper teammate; 4) Position is to maintain the teammate formation for attacking; 5) Intercept is to get the ball as fast as possible; 6) Block is to block the opponent who controls the ball; 7) Trap is to hassle the ball controller and wait to steal the ball; 8) Mark is to mark related opponents; 9) Formation is to maintain formation for defense.

– Attack and Defense: Obviously, the goal of Attack is to attack opponents to score while the goal of Defense is to defense against opponents.
– Root: This is the root task. It firstly evaluate the Attack subtask to see whether it is ready to attack, otherwise it will try the Defense subtask.

The graphical representation of the MAXQ hierarchical structure is shown in Figure 1, where a parenthesis after a subtask's name indicates this subtask will take parameters. It is worth noting that state abstractions are implicitly introduced by this hierarchy. For example in the NavTo subtask, only the agent's own state variables are relevant. It is irrelevant for the KickTo and TackleTo subtasks to consider those state variables describing other players' states. To deal with the large action space, heuristic methods are critical when applying MAXQ-OP. There are many possible candidates depending on the characteristic of subtasks. For instance, hill-climbing is used when searching over the action space of KickTo for the Pass subtask and A* search is used when searching over the action space of dash and turn for the NavTo subtask.

As mentioned earlier, the method for approximating the completion function is crucial for the performance when implementing MAXQ-OP. In RoboCup 2D, it is more challenging to compute the distribution because: 1) the forward search process is unable to run into an sufficient depth due to the online time constraint; and 2) the future states are difficult to predict due to the uncertainty of the environment, especially the unknown behaviors of the opponent team. To estimate the distribution of reaching a goal, we used a variety techniques for different subtasks based on the domain knowledge. Take the Attack subtask for example. A so-called *impelling speed* is used to approximate the completion probability. It is formally defined as:

$$impelling\_speed(s, s', \alpha) = \frac{dist(s, s', \alpha) + pre\_dist(s', \alpha)}{step(s, s') + pre\_step(s')}, \tag{7}$$

where α is a given direction (called aim-angle), $dist(s, s', \alpha)$ is the ball's running distance in direction α from state s to state s', $step(s, s')$ is the estimated steps from state s to state s', $pre\_dist(s')$ estimates final distance in direction α that the ball can be impelled forward starting from state s', and $pre\_step(s')$ estimates the respective steps. The aim-angle in state s is determined dynamically by $aim\_angle(s)$ function. The value of $impelling\_speed(s, s', aim\_angle(s))$ indicates the fact that the faster the ball is moved in a right direction, the more attack chance there would be. In practice, it makes the team attack more efficient. As a result, it can make a fast score within tens of steps in the beginning of a match. Different definitions of the aim_angle function can produce substantially different attack styles, leading to a very flexible and adaptive strategy, particularly for unfamiliar teams.

5 Empirical Evaluation

To test how the MAXQ-OP framework affects our team's final performance, we compared three different versions of our team, including:

– Full: This is exactly the full version of our team, where a complete MAXQ-OP online planning framework is implemented as the key component.

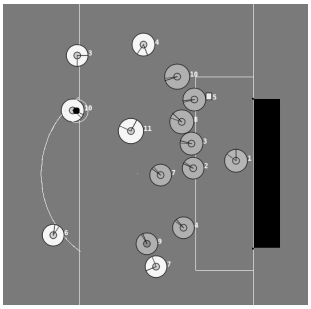

Fig. 2. A selected scene from the final match of RoboCup 2011

- RANDOM: This is nearly the same as FULL, except that when the ball is kickable for the agent and the Shoot behavior finds no solution, the Attack behavior randomly chooses a macro-action to perform between Pass and Dribble with uniform probability.
- HAND-CODED: This is similar to RANDOM, but instead of a random selection between Pass and Dribble, a hand-coded strategy is used. With this strategy, if there is no opponent within 3m from the agent, then Dribble is chosen; otherwise, Pass is chosen.

The only difference between FULL, RANDOM and HAND-CODED is the local selection strategy between Pass and Dribble in the Attack behavior. In FULL, this selection is automatically based on the value function of subtasks (i.e. the solutions found by `EvaluateState(`Pass$,\cdot,\cdot$`)` and `EvaluateState(`Dribble$,\cdot,\cdot$`)` in the MAXQ-OP framework). Although RANDOM and HAND-CODED have different Pass-Dribble selection strategies, the other subtasks of Attack, including Shoot, Pass, Dribble, and Intercept, as that of FULL, remain the same.

For each version, we use an offline coach (also known as a trainer) to independently run the team against the Helios11 binary (which has participated in RoboCup 2011 and won the second place) for 100 episodes. Each episode begins with a fixed scene (i.e. the full state vector) taken from the final match we have participated in of RoboCup 2011, and ends when: 1) our team scores a goal, denoted by **success**; or 2) the ball's x coordination is smaller than -10, denoted by **failure**; or 3) the episode lasts longer than 200 cycles, denoted by **timeout**. It is worth mentioning that although all of the episode begin with the same scene, none of them is identical due to the uncertainty of the environment.

The selected scene, which is originally located at cycle #3142 of that match, is depicted in Figure 2 where white circles represent our players, gray ones represent opponents, and the small black one represents the ball. We can see that our player 10 was holding the ball at that moment, while 9 opponents (including goalie) were blocking just in front of their goal area. In RoboCup 2011, teammate 10 passed the ball directly to teammate 11. Having got the ball, teammate 11 decided to pass the ball back to teammate 10. When teammate 11 had moved to an appropriate position, the ball was passed again to it. Finally, teammate 11 executed a tackle to shoot at cycle #3158 and scored a goal 5 cycles later.

Table 1 summarizes the test results showing that the FULL version of our team outperforms both RANDOM and HAND-CODED with an increase of the

Table 1. Empirical results of WrightEagle in episodic scene test

Version	Episodes	Success	Failure	Timeout
FULL	100	28	31	41
RANDOM	100	15	44	41
HAND-CODED	100	17	38	45

Table 2. Empirical results of WrightEagle in full game test

Opponent Team	Games	Avg. Goals	Avg. Points	Winning Rate
BrainsStomers08	100	3.09 : 0.82	2.59 : 0.28	$82.0 \pm 7.5\%$
Helios10	100	4.30 : 0.88	2.84 : 0.11	$93.0 \pm 5.0\%$
Helios11	100	3.04 : 1.33	2.33 : 0.52	$72.0 \pm 8.8\%$
Oxsy11	100	4.97 : 1.33	2.79 : 0.16	$91.0 \pm 5.6\%$

chance of **sucess** by 86.7% and 64.7% respectively. We find that although FULL, RANDOM and HAND-CODED have the same hierarchical structure and subtasks of Attack, the local selection strategy between Pass and Dribble plays a key role in the decision of Attack and affects the final performance substantially. It can be seen from the table that MAXQ-OP based local selection strategy between Pass and Dribble is sufficient for the Attack behavior to achieve a high performance. Recursively, this is also true for other subtasks over the MAXQ hierarchy, such as Defense, Shoot, Pass, etc. To conclude, MAXQ-OP is able to be the key to success of our team in this episodic scene test.

We also tested the FULL version of our team in full games against 4 best RoboCup 2D opponent teams, namely BrainsStomers08, Helios10, Helios11 and Oxsy11, where BrainStormers08 and Helios10 were the champion of RoboCup 2008 and RoboCup 2010 respectively. In the experiments, we independently ran our team against the binary codes officially released by them for 100 games on exactly the same hardware. Table 2 summarizes the detailed empirical results with our winning rate, which is defined as $p = n/N$, where n is the number of games we won, and N is the total number of games. It can be seen from the table that our team with the implementation of MAXQ-OP substantially outperforms other tested teams. Specifically, our team had about 82.0%, 93.0%, 72.0% and 91.0% of the chances to win BrainsStomers08, Helios10, Helios11 and Oxsy11 respectively.

While there are multiple factors contributing to the general performance of a RoboCup 2D team, it is our observation that our team benefits greatly from the abstraction we made for the actions and states. The key advantage of MAXQ-OP in our team is to provide a formal framework for conducting the search process over a task hierarchy. Therefore, the team can search for a strategy-level solution automatically online by given the pre-defined task hierarchy. To the best of our knowledge, most of the current RoboCup teams develop their team based on hand-coded rules and behaviors.

6 Conclusions

This paper presents a novel approach to automated planning in the RoboCup 2D domain. It benefits from both the advantage of hierarchical decomposition and the power of heuristics. Barry *et al.* proposed an *offline* algorithm called DetH* [3] to solve large MDPs hierarchically by assuming that the transitions between macro-states are totally deterministic. In contrast, we assume a prior distribution over the terminal states of each subtask, which is more realistic. The MAXQ-OP framework has been implemented in the team WrightEagle. The empirical results indicated that the agents developed with this framework and the related techniques reached outstanding performances, showing its potential of scalability to very large domains. This demonstrates the soundness and stability of MAXQ-OP for solving large MDPs with the pre-defined task hierarchy. In the future, we plan to theoretically analyze MAXQ-OP with different task priors and try to generate these priors automatically.

7 Acknowledgments

This work is supported by the National Hi-Tech Project of China under grant 2008AA01Z150 and the Natural Science Foundation of China under grant 60745002 and 61175057. The authors thank Haochong Zhang, Guanghui Lu, and Miao Jiang for their contributions to this work. We are also grateful to the anonymous reviewers for their constructive comments and suggestions.

References

1. Bai, A., Wu, F., Chen, X.: Online planning for large MDPs with MAXQ decomposition (extended abstract). In: Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems. Valencia, Spain (June 2012)
2. Barry, J.: Fast Approximate Hierarchical Solution of MDPs. Ph.D. thesis, Massachusetts Institute of Technology (2009)
3. Barry, J., Kaelbling, L., Lozano-Perez, T.: Deth*: Approximate hierarchical solution of large markov decision processes. In: International Joint Conference on Artificial Intelligence. pp. 1928–1935 (2011)
4. Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of Machine Learning Research 13(1), 63 (May 1999)
5. Gabel, T., Riedmiller, M.: On progress in robocup: the simulation league showcase. RoboCup 2010: Robot Soccer World Cup XIV pp. 36–47 (2011)
6. 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)
7. Riedmiller, M., Gabel, T., Hafner, R., Lange, S.: Reinforcement learning for robot soccer. Autonomous Robots 27(1), 55–73 (2009)
8. Stone, P.: Layered learning in multiagent systems: A winning approach to robotic soccer. The MIT press (2000)
9. Stone, P., Sutton, R., Kuhlmann, G.: Reinforcement learning for robocup soccer keepaway. Adaptive Behavior 13(3), 165–188 (2005)
10. Thrun, S., Fox, D., Burgard, W., Dellaert, F.: Robust monte carlo localization for mobile robots. Artificial intelligence 128(1-2), 99–141 (2001)