Online Planning for Large Markov Decision Processes with Hierarchical Decomposition

AIJUN BAI, FENG WU, and XIAOPING CHEN, University of Science and Technology of China

Markov decision processes (MDPs) provide a rich framework for planning under uncertainty. However, exactly solving a large MDP is usually intractable due to the "curse of dimensionality"— the state space grows exponentially with the number of state variables. Online algorithms tackle this problem by avoiding computing a policy for the entire state space. On the other hand, since online algorithm has to find a near-optimal action online in almost real time, the computation time is often very limited. In the context of reinforcement learning, MAXQ is a value function decomposition method that exploits the underlying structure of the original MDP and decomposes it into a combination of smaller subproblems arranged over a task hierarchy. In this article, we present MAXQ-OP—a novel online planning algorithm for large MDPs that utilizes MAXQ hierarchical decomposition in online settings. Compared to traditional online planning algorithms, MAXQ-OP is able to reach much more deeper states in the search tree with relatively less computation time by exploiting MAXQ hierarchical decomposition online. We empirically evaluate our algorithm in the standard Taxi domain—a common benchmark for MDPs—to show the effectiveness of our approach. We have also conducted a long-term case study in a highly complex simulated soccer domain and developed a team named WrightEagle that has won five world champions and five runners-up in the recent 10 years of RoboCup Soccer Simulation 2D annual competitions. The results in the RoboCup domain confirm the scalability of MAXQ-OP to very large domains.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: MDP, online planning, MAXQ-OP, RoboCup

ACM Reference Format:
Aijun Bai, Feng Wu, and Xiaoping Chen. 2015. Online planning for large Markov decision processes with hierarchical decomposition. ACM Trans. Intell. Syst. Technol. 6, 4, Article 45 (July 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2717316

1. INTRODUCTION

The theory of the Markov decision process (MDP) is very useful for the general problem of planning under uncertainty. Typically, state-of-the-art approaches, such as linear programming, value iteration, and policy iteration, solve MDPs *offline* [Puterman 1994]. In other words, offline algorithms intend to [compute] a policy for the entire state

This work is supported by the National Research Foundation for the Doctoral Program of China under grant 20133402110026, the National Hi-Tech Project of China under grant 2008AA01Z150, and the National Natural Science Foundation of China under grants 60745002 and 61175057.
Authors' addresses: F. Wu and X. Chen, School of Computer Science and Technology, University of Science and Technology of China, Jingzhai Road 96, Hefei, Anhui, 230026, China; emails: {wufeng02, xpchen}@ustc.edu.cn.
Author's current address: A. Bai, EECS Department, UC Berkeley, 750 Sutardja Dai Hall, Berkeley, CA 94720, USA; email: aijunbai@berkeley.edu.

DOI: http://dx.doi.org/10.1145/2717316

space before the agent is actually interacting with the environment. In practice, offline algorithms often suffer from the problem of scalability due to the well-known "curse of dimensionality"—that is, the size of state space grows exponentially with the number of state variables [Littman et al. 1995]. Take, for example, our targeting domain RoboCup Soccer Simulation 2D (RoboCup 2D).[1] Two teams of 22 players play a simulated soccer game in RoboCup 2D. Ignoring some less important state variables (e.g., stamina and view width for each player), the ball state takes four variables, $(x, y, \dot{x}, \dot{y})$, while each player state takes six variables, $(x, y, \dot{x}, \dot{y}, \alpha, \beta)$, where (x, y), $(\dot{x}, \dot{y})$, α, and β are position, velocity, body angle, and neck angle, respectively. Thus, the dimensionality of the resulting state space is $22 \times 6 + 4 = 136$. All state variables are continuous. If we discretize each state variable into only 10^3 values, we obtain a state space containing 10^{408} states. Given such a huge state space, it is prohibitively intractable to solve the entire problem offline. Even worse, the transition model of the problem is subject to change given different opponent teams. Therefore, it is generally impossible to compute a full policy for the RoboCup 2D domain using offline methods.

On the other hand, *online* algorithms alleviate this difficulty by focusing on computing a near-optimal action merely for the current state. The key observation is that an agent can only encounter a fraction of the overall states when interacting with the environment. For example, the total number of timesteps for a match in RoboCup 2D is normally 6,000. Thus, the agent has to make decisions only for those encountered states. Online algorithms evaluate all available actions for the current state and select the seemingly best one by recursively performing forward search over reachable state space. It is worth pointing out that it is not unusual to adopt heuristic techniques in the search process to reduce time and memory usage as in many algorithms that rely on forward search, such as real-time dynamic programming (RTDP) [Barto et al. 1995], LAO* [Hansen and Zilberstein 2001], and UCT [Kocsis and Szepesvári 2006]. Moreover, online algorithms can easily handle unpredictable changes of system dynamics, because in online settings, we only need to tune the decision making for a single timestep instead of the entire state space. This makes them a preferable choice in many real-world applications, including RoboCup 2D. However, the agent must come up with a plan for the current state in almost real time because computation time is usually very limited for online decision making (e.g., only 100ms in RoboCup 2D).

Hierarchical decomposition is another well-known approach to scaling MDP algorithms to large problems. By exploiting the hierarchical structure of a particular domain, it decomposes the overall problem into a set of subproblems that can potentially be solved more easily [Barto and Mahadevan 2003]. In this article, we mainly focus on the method of MAXQ value function decomposition, which decomposes the value function of the original MDP into an additive combination of value functions for sub-MDPs arranged over a task hierarchy [Dietterich 1999a]. MAXQ benefits from several advantages, including temporal abstraction, state abstraction, and subtask sharing. In *temporal abstraction*, temporally extended actions (also known as options, skill, or macroactions) are treated as primitive actions by higher-level subtasks. *State abstraction* aggregates the underlying system states into macrostates by eliminating irrelevant state variables for subtasks. *Subtask sharing* allows the computed policy of one subtask to be reused by some other tasks. For example, in RoboCup 2D, attacking behaviors generally include passing, dribbling, shooting, intercepting, and positioning. Passing, dribbling, and shooting share the same kicking skill, whereas intercepting and positioning utilize the identical moving skill.

In this article, we present MAXQ value function decomposition for online planning (MAXQ-OP), which combines the main advantages of both online planning and MAXQ

[1]http://www.robocup.org/robocup-soccer/simulation/.

hierarchical decomposition to solve large MDPs. Specifically, MAXQ-OP performs online planning to find the near-optimal action for the current state while exploiting the hierarchical structure of the underlying problem. Notice that MAXQ is originally developed for reinforcement learning problems. To the best of our knowledge, MAXQ-OP is the first algorithm that utilizes MAXQ hierarchical decomposition online.

State-of-the-art online algorithms find a near-optimal action for current state via forward search incrementally in depth. However, it is difficult to reach deeper nodes in the search tree within domains with large action space while keeping the appropriate branching factor to a manageable size. For example, it may take thousands of timesteps for the players to reach the goal in RoboCup 2D, especially at the very beginning of a match. Hierarchical decomposition enables the search process to reach deeper states using temporally abstracted subtasks—a sequence of actions that lasts for multiple timesteps. For example, when given a subtask called `moving-to-target`, the agent can continue the search process starting from the target state of `moving-to-target` without considering the detailed plan on specifically moving toward the target, assuming that `moving-to-target` can take care of this. This alleviates the computational burden from searching huge unnecessary parts of the search tree, leading to significant pruning of branching factors. Intuitively, online planning with hierarchical decomposition can cover much deeper areas of the search tree, providing more chance to reach the goal states, and thus potentially improving the action selection strategy to commit a better action for the root node.

One advantage of MAXQ-OP is that we do not need to manually write down complete local policy for each subtask. Instead, we build a MAXQ task hierarchy by defining well the active states, the goal states, and optionally the local-reward functions for all subtasks. Local-reward functions are artificially introduced by the programmer to enable more efficient search processes, as the original rewards defined by the problem may be too sparse to be exploited. Given the task hierarchy, MAXQ-OP automatically finds the near-optimal action for the current state by simultaneously searching over the task hierarchy and building a forward search tree. In the MAXQ framework, a *completion function* for a task gives the expected cumulative reward obtained after finishing a subtask but before completing the task itself following a hierarchical policy. Directly applying MAXQ to online planning requires knowing in advance the completion function for each task following the recursively optimal policy. Thus, obtaining the completion function is equivalent to solving the entire task, which is not applicable in online settings. This poses the major challenge of utilizing MAXQ online.

The key contribution of this article is twofold: the overall framework of exploiting the MAXQ hierarchical structure online and the approximation method made for computing the completion function online. This work significantly extends our previous effort on combining online planning with MAXQ [Bai et al. 2012, 2013b] by introducing a *termination distribution* for each subtask that gives the state distribution when a subtask terminates and proposing a new method to approximate termination distributions. The experimental results in the standard Taxi domain—a common benchmark for MDPs—confirm the efficiency and effectiveness of MAXQ-OP with the new approximation method. Most importantly, we present our long-term case study in RoboCup 2D by deploying MAXQ-OP and developing a team of autonomous agents, namely WrightEagle.[2] Our team has participated in annual RoboCup competitions since 1999, winning five world championships and named runner-up five times in the past 10 years. The experimental and competition results show that MAXQ-OP can scale to very large problems with outstanding performance.

[2]http://wrighteagle.org/2d/.

The remainder of this article is organized as follows. Section 2 introduces the literature related to our work. Section 3 briefly reviews the background of MDP and MAXQ. Section 4 describes in detail our main algorithm—MAXQ-OP. Section 5 discusses how the MAXQ hierarchical structure can be derived, the state abstraction in MAXQ-OP algorithm, and some advantages and drawbacks of MAXQ-OP algorithm compared to traditional online planning algorithms. Section 6 reports our experimental results in the Taxi domain, and Section 7 presents the case study in RoboCup 2D domain. In Section 8, we conclude with discussion on potential future work.

2. RELATED WORK

In the context of online planning for MDPs, RTDP [Barto et al. 1995; Bonet and Geffner 2003; McMahan et al. 2005; Sanner et al. 2009] is among the first that tries to find a near-optimal action for the current state by conducting a trial-based search process with greedy action selection and an admissible heuristic. Instead of trial-based search, AO* [Hansen and Zilberstein 2001; Feng and Hansen 2002; Bonet and Geffner 2012] builds an optimal solution graph with respect to the AND-OR graph by greedily expanding tip nodes in the current best partial solution graph and assigning values to new nodes according to an admissible heuristic function. Monte Carlo tree search (MCTS) [Kearns et al. 1999; Kocsis and Szepesvári 2006; Gelly and Silver 2011; Browne et al. 2012; Feldman and Domshlak 2012; Bai et al. 2013a] finds near-optimal policies by combining tree search methods with Monte Carlo sampling techniques. The key idea is to evaluate each state in a best-first search tree using simulation samples. Most recently, trial-based heuristic tree search (THTS) [Keller and Helmert 2013] is proposed to subsume these approaches by classifying five ingredients: heuristic function, backup function, action selection, outcome selection, and trial length. Although they all try to find a near-optimal action online for the current state, they do not exploit the underlying hierarchical structure of the problem as our approach—MAXQ-OP.

In the research of reinforcement learning, hierarchical decomposition has been adopted under the name of hierarchical reinforcement learning (HRL) [Barto and Mahadevan 2003]. HRL aims to learn a policy for an MDP efficiently by exploiting the underlying structure while interacting with the environment. One common approach is using state abstraction to partition the state space into a set of subspaces, namely macrostates, by eliminating irrelevant state variables [Andre and Russell 2002; Asadi and Huber 2004; Hengst 2004; Manfredi and Mahadevan 2005; Li et al. 2006; Bakker et al. 2005; Hengst 2007]. In particular, Sutton et al. [1999] model HRL as a semi-Markov decision process (SMDP) by introducing temporally extended actions, namely options. Each option is associated with an inner policy that can be either manually specified or learned by the agent. Our work is based on the MAXQ value function decomposition originally proposed by Dietterich [1999a] in the context of HRL. MAXQ-based HRL methods convert the original MDP into a hierarchy of SMDPs and learn the solutions simultaneously [Diuk et al. 2006; Jong and Stone 2008].

Similar to reinforcement learning, there exist several offline MDP planning algorithms that also exploit the hierarchical structure to speed up the planning process. For instance, Hauskrecht et al. [1998] develop an *abstract MDP* model that works with macroactions and macrostates by treating macroactions as local policies that act in certain regions of state space and restricting states in the abstract MDP to those at the boundaries of regions. Variable influence structure analysis (VISA) [Jonsson and Barto 2006] performs hierarchical decomposition for an MDP by building dynamic Bayesian network (DBN) models for actions, and constructing causal graphs that capture relationships between state variables, under the assumption that a factored MDP model is available. Barry et al. [2011] propose an offline algorithm called *DetH** to solve

large MDPs hierarchically by assuming that the transitions between macrostates are deterministic.

Although hierarchical decomposition has been widely used in the literature of reinforcement learning and offline planning for MDPs, it is still nontrivial when applying it in online settings. The key challenge is that when searching with high-level actions (tasks), it is critical to know how they can be fulfilled by low-level actions (subtasks or primitive actions). For example, in the robot soccer domain, if a player wants to shoot the goal (high-level action), it must first know how to adjust its position and kick the ball toward a specified position (low-level actions). Unfortunately, this information is not available in advance during online planning. As aforementioned, we address this challenge by introducing a *termination distribution* for each subtask over its terminal states and assuming that subtasks will take care of the local policies to achieve the termination distributions. More detail will be described in Section 4.4.

3. BACKGROUND

In this section, we briefly review the MDP model [Puterman 1994] and the MAXQ hierarchical decomposition method [Dietterich 1999a].

3.1. MDP Framework

Formally, an MDP is defined as a 4-tuple $\langle S, A, T, R\rangle$, where

—S is a set of states;
—A is a set of actions;
—$T : S \times A \times S \to [0, 1]$ is the transition function, with $T(s' \mid s, a) = \Pr(s' \mid s, a)$ denoting the probability of reaching state s' after action a is performed in state s; and
—$R : S \times A \to \mathbb{R}$ is the reward function, with $R(s, a)$ denoting the immediate reward obtained by applying action a in state s.

A *policy* defined for an MDP is a mapping from states to actions $\pi : S \to A$, with $\pi(s)$ denoting the action to take in state s. The *value function* $V^\pi(s)$ of a policy π is defined as the expected cumulative reward by following policy π starting from state s:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right], \tag{1}$$

where $\gamma \in (0, 1]$ is a discount factor. The *action value function* $Q^\pi(s, a)$ is defined as the expected cumulative reward by first performing action a in state s and following π thereafter:

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s' \mid s, a) V^\pi(s'). \tag{2}$$

Solving an MDP is equivalent to finding the *optimal* policy π^* such that for any policy π and any state $s \in S$, $V^{\pi^*}(s) \geq V^\pi(s)$ holds. The optimal value functions $V^{\pi^*}(s)$ and $Q^{\pi^*}(s, a)$ (we denote them as $V^*(s)$ and $Q^*(s, a)$ for short) satisfy the well-known Bellman equations [Bellman 1957]:

$$V^*(s) = \max_{a \in A} Q^*(s, a). \tag{3}$$

Given the optimal value functions by solving the Bellman equations, the optimal policy π^* can then be obtained by using

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s, a). \tag{4}$$

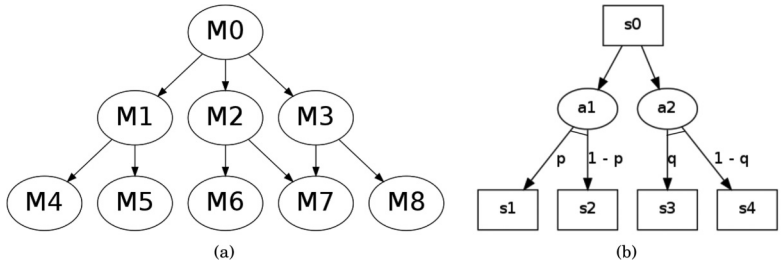


Fig. 1. An example of MAXQ task graph (a) and an example of MDP AND-OR tree (b).

In this work, we assume that there exists a set of goal states $G \subseteq S$ such that for all $g \in G$ and $a \in A$, we have $T(g \mid g, a) = 1$ and $R(g, a) = 0$. If the discount factor $\gamma = 1$, the resulting MDP is then called *undiscounted goal-directed MDP* (a.k.a. *stochastic shortest path* problem [Bertsekas 1996]). It has been proved that any MDP can be transformed into an equivalent undiscounted negative-reward goal-directed MDP where the reward for nongoal states is strictly negative [Barry 2009]. Hence, undiscounted goal-directed MDP is actually a general formulation. Here, we are focusing on undiscounted goal-directed MDPs. However, our algorithm and results can be straightforwardly applied to other equivalent models.

3.2. MAXQ Hierarchical Decomposition

The MAXQ hierarchical decomposition method decomposes the original MDP M into a set of sub-MDPs arranged over a hierarchical structure [Dietterich 1999a]. Each sub-MDP is treated as an macroaction for high-level MDPs. Specifically, let the decomposed MDPs be $\{M_0, M_1, \ldots, M_n\}$, then M_0 is the root subtask such that solving M_0 solves the original MDP M. Each subtask M_i is defined as a tuple $\langle \tau_i, A_i, \tilde{R}_i \rangle$, where

—τ_i is the *termination predicate* that partitions the state space into a set of active states S_i and a set of terminal states G_i (also known as subgoals);
—A_i is a set of (macro)actions that can be selected by M_i, which can either be primitive actions of the original MDP M or low-level subtasks; and
—$\tilde{R}_i$ is the (optional) *local-reward* function that specifies the rewards for transitions from active states S_i to terminal states G_i.

A subtask can also take parameters, in which case different bindings of parameters 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. This hierarchical structure can be represented as a directed acyclic graph—the *task graph*. An example of task graph is shown in Figure 1(a). In the figure, root task M_0 has three macroactions: M_1, M_2, and M_3 (i.e., $A_0 = \{M_1, M_2, M_3\}$). Subtasks M_1, M_2, and M_3 are sharing lower-level primitive actions M_i ($4 \leq i \leq 8$) as their subtasks. In other words, a subtask in the task graph is also a (macro)action of its parent. Each subtask must be fulfilled by a policy unless it is a primitive action.

Given the hierarchical structure, a *hierarchical policy* π is defined as a set of policies for each subtask $\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$, where π_i for subtask M_i is a mapping from its active states to actions $\pi_i : S_i \rightarrow A_i$. The *projected value function* $V^\pi(i, s)$ is defined as the expected cumulative reward of following a hierarchical policy $\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$ starting from state s until M_i terminates at one of its terminal states $g \in G_i$. Similarly, the action value function $Q^\pi(i, s, a)$ for subtask M_i is defined as the expected cumulative reward of first performing action M_a (which is also a subtask) in state s, then following

policy π until the termination of M_i. Notice that for primitive subtasks M_a, we have $V^\pi(a, s) = R(s, a)$.

It has been shown that the value functions of a hierarchical policy π can be expressed recursively as follows [Dietterich 1999a]:

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a), \tag{5}$$

where

$$V^\pi(i, s) = \begin{cases} R(s, i), & \text{if } M_i \text{ is primitive} \\ Q^\pi(i, s, \pi(s)). & \text{otherwise} \end{cases} \tag{6}$$

Here, $C^\pi(i, s, a)$ is the *completion function* that specifies the expected cumulative reward obtained after finishing subtask M_a but before completing M_i when following the hierarchical policy π, defined as

$$C^\pi(i, s, a) = \sum_{s' \in G_i, N \in \mathbb{N}^+} \gamma^N \Pr(s', N \mid s, a) V^\pi(i, s'), \tag{7}$$

where $\Pr(s', N \mid s, a)$ is the probability that subtask M_a will terminate in state s' after N timesteps of execution. A *recursively optimal policy* π^* can be found by recursively computing the optimal projected value function as

$$Q^*(i, s, a) = V^*(a, s) + C^*(i, s, a), \tag{8}$$

where

$$V^*(i, s) = \begin{cases} R(s, i), & \text{if } M_i \text{ is primitive} \\ \max_{a \in A_i} Q^*(i, s, a). & \text{otherwise} \end{cases} \tag{9}$$

In Equation (8), $C^*(i, s, a) = C^{\pi^*}(i, s, a)$, is the completion function of the recursively optimal policy π^*. Given the optimal value functions, the optimal policy π_i^* for subtask M_i is then given as

$$\pi_i^*(s) = \operatorname*{argmax}_{a \in A_i} Q^*(i, s, a). \tag{10}$$

4. ONLINE PLANNING WITH MAXQ

In general, online planning interleaves planning with execution and chooses a near-optimal action only for the current state. Given the MAXQ hierarchy of an MDP (i.e., $M = \{M_0, M_1, \ldots, M_n\}$), the main procedure of MAXQ-OP evaluates each subtask by forward searching and computing 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 primitive subtasks at the leaf nodes. After that, the best action $a \in A_0$ is selected for the root task M_0 based on the resulting action values. Accordingly, a primitive action $a_p \in A$ that should be performed first is also determined. By performing a_p, the environment transits to a new state. Then, the planning procedure repeats by selecting the seemingly best action for the new time step. The basic idea of MAXQ-OP is to approximate Equation (8) online. The main challenge is the approximation of completion function. Section 4.1 gives an overview of the MAXQ-OP algorithm before presenting it in detail.

4.1. Overview of MAXQ-OP

The key challenge of MAXQ-OP is to estimate the value of the completion function. Intuitively, the completion function represents the optimal value obtained from fulfilling the task M_i after executing a subtask M_a, but before completing task M_i. According to

Equation (7), the completion function of the optimal policy π^* is written as

$$C^*(i,s,a) = \sum_{s' \in G_i, N \in \mathbb{N}^+} \gamma^N \Pr(s', N \mid s, a) V^*(i, s'), \tag{11}$$

where

$$\begin{aligned} \Pr(s', N \mid s, a) = \textstyle\sum_{\langle s, s_1, \ldots, s_{N-1} \rangle} \; & T(s_1 \mid s, \pi_a^*(s)) \cdot T(s_2 \mid s_1, \pi_a^*(s_1)) \\ & \ldots T(s' \mid s_{N-1}, \pi_a^*(s_{N-1})) \Pr(N \mid s, a), \end{aligned} \tag{12}$$

where $T(s' \mid s, a)$ is the transition function of the underlying MDP and $\Pr(N \mid s, a)$ is the probability that subtask M_a will terminate in N steps starting from state s. Here, $\langle s, s_1, \ldots, s_{N-1} \rangle$ is a length-N path from state s to the terminal state s' by following the local optimal policy $\pi_a^* \in \pi^*$. Unfortunately, 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. However, this is inapplicable to online algorithms, especially for large domains.

To exactly compute the optimal completion function $C^*(i, s, a)$, the agent must know the optimal policy π^*. As mentioned, this is equivalent to solving the entire problem. Additionally, it is intractable to find the optimal policy online due to time constraints. When applying MAXQ to online algorithms, approximation is necessary to compute the completion function for each subtask. One possible solution is to calculate an approximate policy offline and then 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 when the domain is very large.

In the MAXQ framework, given an optimal policy, a subtask terminates in any goal state with probability 1 after several timesteps of execution. Notice that the term γ^N in Equation (7) is equal to 1, as we are focusing on problems with goal states and in our settings the γ value is assumed to be exactly 1. The completion function can then be rewritten as

$$C^*(i,s,a) = \sum_{s' \in G_i} P_t(s' \mid s, a) V^*(i, s'), \tag{13}$$

where $P_t(s' \mid s, a) = \sum_N \Pr(s', N \mid s, a)$ is a marginal distribution defined over the terminal states of subtask M_i, giving the probability that subtask M_a will terminate at state s' starting from state s. Therefore, to estimate the completion function, we need to first estimate $P_t(s' \mid s, a)$, which we call the *termination distribution*. Thus, for nonprimitive subtasks, according to Equation (9), we have

$$V^*(i,s) \approx \max_{a \in A_i} \left\{ V^*(a,s) + \sum_{s' \in G_a} \Pr(s' \mid s, a) V^*(i, s') \right\}. \tag{14}$$

Although Equation (14) implies the approximation of completion function, it is still inapplicable to compute online, as Equation (14) is recursively defined over itself. To this end, we introduce depth array d and maximal search depth array D, where $d[i]$ is current search depth in terms of macroactions for subtask M_i and $D[i]$ gives the maximal allowed search depth for subtask M_i. A heuristic function H is also introduced to estimate the value function when exceeding the maximal search depth. Equation (14) is then approximated as

$$V(i,s,d) \approx \begin{cases} H(i,s), & \text{if } d[i] \geq D[i] \\ \max_{a \in A_i} \{ V(a,s,d) + \\ \sum_{s' \in G_a} \Pr(s' \mid s, a) V(i, s', d[i] \leftarrow d[i]+1) \}, & \text{otherwise.} \end{cases} \tag{15}$$

Equation (15) gives the overall framework of MAXQ-OP, which makes applying MAXQ online possible. When implementing the algorithm, calling $V(0, s, [0, 0, \ldots, 0])$ returns the value of state s in root task M_0, as well as a primitive action to be performed by the agent.

In practice, instead of evaluating all terminal states of a subtask, we sample a subset of terminal states. Let $G_{s,a} = \{s' \mid s' \sim P_t(s' \mid s, a)\}$ be the set of sampled states; the completion function is then approximated as

$$C^*(i, s, a) \approx \frac{1}{|G_{s,a}|} \sum_{s' \in G_{s,a}} V^*(i, s'). \tag{16}$$

Furthermore, Equation (15) can be rewritten as

$$V(i, s, d) \approx \begin{cases} H(i, s), & \text{if } d[i] \geq D[i] \\ \max_{a \in A_i} \{V(a, s, d) + \\ \sum_{s' \in G_{s,a}} \frac{1}{|G_{s,a}|} V(i, s', d[i] \leftarrow d[i] + 1)\}, & \text{otherwise.} \end{cases} \tag{17}$$

It is worth noting that Equation (17) is introduced to prevent enumerating the entire space of terminal states of a subtask, which could be huge.

ALGORITHM 1: `OnlinePlanning()`

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

Let $r \leftarrow 0$;
Let $s \leftarrow$ `GetInitState()`;
Let *root_task* $\leftarrow 0$;
Let *depth_array* $\leftarrow [0, 0, \ldots, 0]$;

while $s \notin G_0$ **do**
  $\langle v, a_p \rangle \leftarrow$ `EvaluateStateInSubtask`(*root_task*, s, *depth_array*);
  $r \leftarrow r +$ `ExecuteAction`(a_p, s);
  $s \leftarrow$ `GetNextState()`;

return r;

4.2. Main Procedure of MAXQ-OP

The overall process of MAXQ-OP is shown in Algorithm 1, where state s is initialized by `GetInitState` function and `GetNextState` function returns the next state of the environment after `ExecuteAction` function is executed. The main process loops over until a goal state in G_0 is reached. Notice that the key procedure of MAXQ-OP is `EvaluateStateInSubtask`, which evaluates each subtask by depth-first search and returns the seemingly best action for the current state. `EvaluateStateInSubtask` function is called with a depth array containing all zeros for all subtasks. Section 4.3 explains `EvaluateStateInSubtask` function in detail.

4.3. Task Evaluation over Hierarchy

To choose a near-optimal action, an agent must compute the action value function for each available action in current state s. Typically, this process builds a search tree starting from s and ending with one of 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 outcomes of action activation(i.e., states in MDP settings) [Nilsson 1982; Hansen and Zilberstein 2001]. The root node of such an AND-OR tree represents the current state. The search in

ALGORITHM 2: `EvaluateStateInSubtask`(i, s, d)

Input: subtask M_i, state s and depth array d
Output: $\langle V^*(i,s)$, 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
    Let ⟨v*, a_p*⟩ ← ⟨−∞, nil⟩;
    for M_k ∈ Subtasks(M_i) do
        if M_k is primitive or s ∉ G_k then
            Let ⟨v, a_p⟩ ← EvaluateStateInSubtask(k, s, d);
            v ← v+ EvaluateCompletionInSubtask(i, s, k, d);
            if v > v* then
                ⟨v*, a_p*⟩ ← ⟨v, a_p⟩;
    return ⟨v*, a_p*⟩;
```

the tree is proceeded in a best-first manner until a goal state or a maximal search depth is reached. When reaching the maximal depth, a heuristic function is usually used to estimate the expected cumulative reward for the remaining timesteps. Figure 1(b) gives an example of the AND-OR tree. In the figure, s_0 is the current state with two actions a_1 and a_2 available for s_0. The corresponding transition probabilities are $T(s_1 \mid s_0, a_1) = p$, $T(s_2 \mid s_0, a_1) = 1 - p$, $T(s_3 \mid s_0, a_2) = q$, and $T(s_4 \mid s_0, a_2) = 1 - q$.

In the presence of a task hierarchy, Algorithm 2 summarizes the pseudocode of the search process of MAXQ-OP. MAXQ-OP expands the node of the current state s by recursively evaluating each subtask of M_i, estimates the respective completion function, and finally selects the subtask with the highest returned value. The recursion terminates when (1) the subtask is a primitive action; (2) the state is a goal state or a state beyond the scope of this subtask's active states; or (3) the maximal search depth is reached—that is, $d[i] \geq D[i]$. Note that each subtask can have different maximal depths (e.g., subtasks in the higher level may have smaller maximal depth in terms of evaluated macroactions). If a subtask corresponds to a primitive action, an immediate reward will be returned together with the action. If the search process exceeds the maximal search depth, a heuristic value is used to estimate the future long-term reward. In this case, a *nil* action is also returned (however, it will not be chosen by high-level subtasks in the algorithm's implementation). In other cases, `EvaluateStateInSubtask` function recursively evaluates all lower-level subtasks and finds the seemingly best (macro)action.

4.4. Completion Function Approximation

As shown in Algorithm 3, a recursive procedure is developed to estimate the completion function according to Equation (17). Here, termination distributions need to be provided for all subtasks in advance. Given a subtask with domain knowledge, it is possible to approximate the respective termination distribution either offline or online. For subtasks with few goal states, such as robot navigation or manipulation, offline approximation is possible—it is rather reasonable to assume that these subtasks will terminate when reaching any of the target states; for subtasks that have a wide range of goal states, either desired target states or just failures for the subtasks, such as passing the ball to a teammate or shooting the ball in presence of opponents in RoboCup 2D, online approximation is preferable given some assumptions of the transition model.

ALGORITHM 3: `EvaluateCompletionInSubtask`(i, s, a, d)

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

Let $G_{s,a} \leftarrow \{s' \mid s' \sim P_t(s' \mid s, a)\}$;
Let $v \leftarrow 0$;
for $s' \in G_{s,a}$ **do**
  $d' \leftarrow d$;
  $d'[i] \leftarrow d'[i] + 1$;
  $v \leftarrow v +$ `EvaluateStateInSubtask`(i, s', d');
$v \leftarrow \frac{v}{|G_{s,a}|}$;
return v;

ALGORITHM 4: `NextAction`(i, s)

Input: subtask index i and state s
Output: selected action a^*

if `SearchStopped`(i, s) **then**
  return nil;
else
  Let $a^* \leftarrow \arg\max_{a \in A_i} H_i[s, a] + c\sqrt{\frac{\ln N_i[s]}{N_i[s,a]}}$;
  $N_i[s] \leftarrow N_i[s] + 1$;
  $N_i[s, a^*] \leftarrow N_i[s, a^*] + 1$;
  return a^*;

Notice that a goal state for a subtask is a state where the subtask terminates, which could be a successful situation for the subtask but could also be a failed situation. For example, when passing the ball to a teammate, the goal states are the cases in which either the ball is successfully passed, the ball has been intercepted by any of the opponents, or the ball is running out of the field. Although it is not mentioned in the algorithm, it is also possible to cluster the goal states into a set of classes (e.g., success and failure), sample or pick a representative state for each class, and use the representatives to recursively evaluate the completion function. This clustering technique is very useful for approximating the completion functions for subtasks with huge numbers of goal states. Take RoboCup 2D, for example. The terminating distributions for subtasks such as pass, intercept, and dribble usually have several peaks for the probability values. Intuitively, each peak corresponds to a representative state that is more likely to happen than others. Instead of sampling from the complete terminating distribution, we use these representative states to approximate the completion function. Although this is only an approximate for the real value, it is still very useful for action selection in the planning process. How to theoretically bound the approximation error will be a very interesting challenge but is beyond the scope of this work.

4.5. Heuristic Search in Action Space

For domains with large action space, it may be very time consuming to enumerate all possible actions (subtasks). Hence, it is necessary to use heuristic techniques (including pruning strategies) to speed up the search process. Intuitively, there is no need to evaluate those actions that are not likely to be better than currently evaluated actions. In MAXQ-OP, this is done by implementing an iterative version of `Subtasks` function using a `NextAction` procedure to dynamically select the most promising action to be

evaluated next with the trade-off between exploitation and exploration. The trade-off between exploitation and exploration is needed because the agent does not know the particular order in terms of action values among (macro)actions for the current evaluated state before the complete search (otherwise, the agent does not have to search), in which case the agent should not only exploit by evaluating the seemingly good actions first but also should explore other actions for higher future payoffs.

Different heuristic techniques, such as A*, hill-climbing, and gradient ascent, can be used for different subtasks. Each of them may have a different heuristic function. However, these heuristic values do not need to be comparable to each other, as they are only used to suggest the next action to be evaluated for the specific subtask. In other words, the heuristic function designed for one subtask is not used for the other subtasks during action selection. Once the search terminates, only the chosen action is returned. Therefore, different heuristic techniques are only used inside `NextAction`. However, for each subtask, the heuristic function (as `HeuristicValue` in Algorithm 2) is designed to be globally comparable because it is used by all subtasks to give an estimation of the action evaluation when the search reaches the maximal search depth.

For large problems, a complete search in the state space of a subtask is usually intractable even if we have the explicit representation of the system dynamics available. To address this, we use a Monte Carlo method as shown in Algorithm 4, where the UCB1 [Auer et al. 2002] version of `NextAction` function is defined. By so doing, we do not have to perform a complete search in the state space to select an action, as only visited states in the Monte Carlo search tree are considered. Additionally, the algorithm has the very nice anytime feature that is desirable for online planning because the planning time is very limited. It is worth noting that for Monte Carlo methods, the exploration strategy is critical to achieve good performance. Therefore, we adopt the UCB1 method, which guarantees convergence to the optimal solution given sufficient amount of simulations [Auer et al. 2002]. Furthermore, it has been shown to be very useful for exploration in a large solution space [Gelly and Silver 2011].

Here, in Algorithm 4, $N_i[s]$ and $N_i[s,a]$ are the visiting counts of state s and state-action pair (s,a), respectively, for subtask M_i, and $c\sqrt{\ln N_i[s]/N_i[s,a]}$ is a biased bonus with higher value for rarely tried actions to encourage exploration on them, where c is a constant variable that balances the trade-off between exploitation and exploration. These values are maintained and reused during the whole process when the agent is interacting with the environment. The procedure `SearchStopped` dynamically determines whether the search process for the current task should be terminated based on pruning conditions (e.g., the maximal number of evaluated actions, or the action-value threshold). $H_i[s,a]$ are heuristic values of applying action a in state s for subtask M_i, initialized according to domain knowledge. They can also be updated incrementally while the agent interacts with the environment, for example, according to a learning rule, $H_i[s,a] \leftarrow (1-\alpha)H_i[s,a] + \alpha Q(i,s,a)$, which is commonly used in reinforcement learning algorithms [Sutton and Barto 1998].

5. DISCUSSION: MAXQ-OP ALGORITHM

In this article, the MAXQ task hierarchy used in the MAXQ-OP algorithm is assumed to be provided by the programmer according to some prior domain knowledge. In other words, the programmer needs to identify subgoals in the underlying problem and define subtasks that achieve these subgoals. For example, in the RoboCup 2D domain, this requires the programmer to have some knowledge about the soccer game and be able to come up with some subtasks, including shooting, dribbling, passing, positioning, and so forth. Given the hierarchical structure, MAXQ-OP automatically searches over the

task structure, as well as the state space, to find out the seemingly best action for the current state, taking advantage of some specified heuristic techniques.

Another promising approach that has been drawing much research interest is discovering the hierarchical structure automatically from state-action histories in the environment, either online or offline [Hengst 2002; Stolle 2004; Bakker and Schmidhuber 2004; Şimşek et al. 2005; Mehta et al. 2008, 2011]. For example, Mehta et al. [2008] presents hierarchy induction via models and trajectories (HI-MAT), which discovers MAXQ task hierarchies by applying DBN models to successful execution trajectories of a source MDP task; the HEXQ [Hengst 2002, 2004] method decomposes MDPs by finding nested sub-MDPs where there are policies to reach any exit with certainty; and Stolle [2004] performs automatic hierarchical decomposition by taking advantage of the factored representation of the underlying problem. The resulting hierarchical structure discovered by these methods can be directly used to construct a MAXQ task graph, which can then be used to implement the MAXQ-OP algorithm. The combined method is automatically applicable to general domains.

One important advantage of MAXQ-OP algorithm is that it is able to transfer the MAXQ hierarchical structure from one domain to other similar domains [Mehta et al. 2008; Taylor and Stone 2009]. Transferring only structural knowledge across MDPs is shown to be a viable alternative to transferring the entire value function or learned policy itself, which can also be easily generalized to similar problems. For example, in the eTaxi domain, the same MAXQ structure can be used without modifications for problem instances with different sizes. This also provides a possibility to discover or design a MAXQ hierarchical structure for smaller problems, then transfer it to larger problems to be reused. With techniques of designing, discovering, and transferring MAXQ hierarchical structural, the MAXQ-OP algorithm is applicable to a wide range of problems.

Another advantage of the MAXQ hierarchical structure is the ability to exploit state abstractions so that individual MDPs within the hierarchy can ignore large parts of the state space [Dietterich 1999b]. Each action in the hierarchy abstracts away irrelevant state variables without compromising the resulting online policy. For a subtask, a set of state valuables Y can be abstracted if the joint transition function for each child action can be divided into two parts, where the part related to Y does not affect the probability of execution for a certain number of steps—for instance,

$$\Pr(x', y', N \mid x, y, a) = \Pr(x', N \mid x, a) \times \Pr(y' \mid y, a), \tag{18}$$

where x and x' give values for state variables in X; y and y' give values for state variables in Y; $X \cup Y$ is the full state vector; and a is a child action, which could be either a macroaction or a primitive action. For example, in RoboCup 2D, if the agent is planning for the best action to move to a target position from its current position as fast as possible, then the state variables representing the ball and other players are irrelevant for the moving subtask. For a primitive action, those state variables that do not affect the primitive transition and reward models can be abstracted away. As an example, in RoboCup 2D, the positions of other players are irrelevant to kick action given the relative position of the ball, because the kick action has the same transition and reward models despite the location of other players. By ignoring irrelevant state variables during the search processes for subtasks, state abstractions make the algorithm more efficient when searching over the state space, as a state in its abstracted form actually represents a subspace of the original state space. Evaluating an abstracted state is actually evaluating a set of states in the original state space. In MAXQ-OP, state abstractions are assumed to be provided for all subtasks together with the MAXQ hierarchy, according to the domain knowledge of the underlying problem.

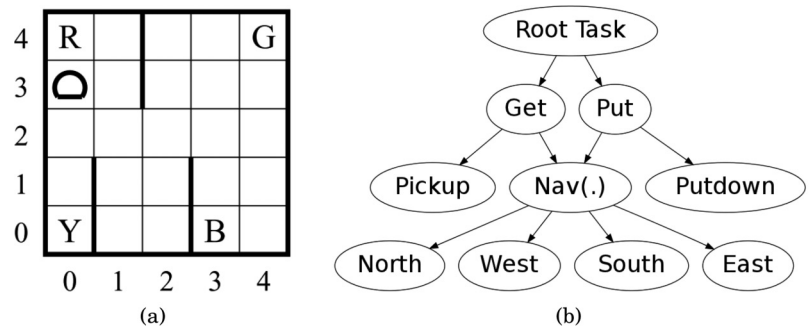


Fig. 2. The Taxi domain (a) and the MAXQ task graph for Taxi (b).

Compared to traditional online planning algorithms, the success of MAXQ-OP is due to the fact that it is able to reach much deeper nodes in the search tree by exploiting hierarchical structure given the same computation resources. Traditional online planning algorithms, such as RTDP, AOT, and MCTS, search only in state space, by step-by-step expanding the search node to recursively evaluate an action at the root node. The search process terminates at a certain depth with the help of a heuristic function that assumes the goal state has been reached. A typical search path of this search process can be summarized in Equation (19), where s_i is the state node, s_H is the deepest state node where a heuristic function is called, $\rightarrow$ is the state transition, $\rightsquigarrow$ represents the calling of a heuristic function, and g is one of the goal states:

$$[s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \rightarrow s_H] \rightsquigarrow g. \tag{19}$$

The MAXQ-OP algorithm searches not only in the state space but also over the task hierarchy. For each subtask, only a few steps of macroactions are searched. The remaining steps are abstracted away by using a heuristic function inside the subtask, and a new search will be invoked at one of the goal states of previous searched subtasks. This leads to a large number of prunings in the state space. A search path of running MAXQ-OP for a MAXQ task graph with two levels of macroactions (including root task) is summarized in Equation (20), where s_{H_i} is the deepest searched state node in one subtask; g_i is one of the goal states of a subtask, which is also a start state for another subtask; and g is one of the goal states for the root task:

$$[s_1 \rightarrow \cdots \rightarrow s_{H_1}] \rightsquigarrow [g_1/s'_1 \rightarrow \cdots \rightarrow s'_{H_2}] \rightsquigarrow [g_2/s''_1 \rightarrow \cdots \rightarrow s''_{H3}] \cdots \rightsquigarrow g. \tag{20}$$

One drawback of MAXQ-OP is the significant amount of domain knowledge that must be adopted for the algorithm to work well. More specifically, constructing the hierarchy, incorporating heuristic techniques for subtasks, and estimating the termination distributions either online or offline require domain knowledge to work well. For complex problems, this will not be an effort that can be ignored. On the other hand, automatically solving highly complicated problems with huge state and action spaces is quite challenging. The ability to exploit various domain knowledge to enhance the solution quality for complex problems can also be seen as one advantage of the MAXQ-OP method.

6. EXPERIMENTS: THE TAXI DOMAIN

The standard Taxi domain is a common benchmark problem for hierarchical planning and learning in MDPs [Dietterich 1999a]. As shown in Figure 2(a), it consists of a 5×5 grid world with walls and 4 taxi terminals: R, G, Y, and B. The goal of a taxi agent is to pick up and deliver a passenger. The system has four state variables:

Table I. Complete Definitions of Nonprimitive Subtasks for the Taxi Domain

Subtask	Active States	Terminal States	Actions	Max Depth
Root	All states	$pl = dl$	Get and Put	2
Get	$pl \neq taxi$	$pl = taxi$	Nav(t) and Pickup	2
Put	$pl = taxi$	$pl = dl$	Nav(t) and Putdown	2
Nav(t)	All states	$(x, y) = t$	North, South, East, and West	7

the agent's coordination x and y, the pickup location pl, and the destination dl. The variable pl can be one of the 4 terminals, or just *taxi* if the passenger is inside the taxi. The variable dl must be one of the 4 terminals. In our experiments, pl is not allowed to equal dl. Therefore, this problem has totally 404 states with 25 taxi locations, 5 passenger locations, and 4 destination locations, excluding the states where $pl = dl$. This is identical to the setting of Jong and Stone [2008]. At the beginning of each episode, the taxi's location, the passenger's location, and the passenger's destination are all randomly generated. The problem terminates when the taxi agent successfully delivers the passenger. There are six primitive actions: (a) four navigation actions that move the agent into one neighbor grid—North, South, East, and West; (b) the Pickup action; and (c) the Putdown action. Each navigation action has a probability of 0.8 to successfully move the agent in the desired direction and a probability of 0.1 for each perpendicular direction. Each legal action has a reward of -1, whereas illegal Pickup and Putdown actions have a penalty of -10. The agent also receives a final reward of $+20$ when the episode terminates with a successful Putdown action.

When applying MAXQ-OP in this domain, we use the same MAXQ hierarchical structure proposed by Dietterich [1999a], as shown in Figure 2(b). Note that the Nav(t) subtask takes a parameter t, which could either be R, G, Y, or B, indicating the navigation target. In the hierarchy, the four primitive actions and the four navigational actions abstract away the passenger and destination state variables. Get and Pickup ignore destination, and Put and Putdown ignore passenger. The definitions of the nonprimitive subtasks are shown in Table I. The Active States and Terminal States columns give the active and terminal states for each subtask, respectively; the Actions column gives the child (macro)actions for each subtask; and the Max Depth column specifies the maximal forward search depths in terms of (macro)actions allowed for each subtask in the experiments.

The procedure `EvaluateCompletionInSubtask` is implemented as follows. For high-level subtasks such as Root, Get, Put, and Nav(t), we assume that they will terminate in the designed goal states with probability 1, and for primitive subtasks such as North, South, East, and West, the domain's underlying transition model $T(s' \mid s, a)$ is used to sample a next state according to its transition probability. For each nonprimitive subtask, the function `HeuristicValue` is designed as the sum of the negative of a Manhattan distance from the taxi's current location to the terminal state's location and other potential immediate rewards. For example, the heuristic value for the Get subtask is defined as $-\texttt{Manhattan}((x, y), pl) - 1$, where $\texttt{Manhattan}((x_1, y_1), (x_2, y_2))$ gives the Manhattan distance $|x_1 - x_2| + |y_1 - y_2|$.

A cache-based pruning strategy is implemented to enable more effective subtask sharing. More precisely, if state s has been evaluated for subtask M_i with depth $d[i] = 0$, suppose that the result is $\langle v, a_p \rangle$; then, this result will be stored in a cache table as

$$cache[i, hash(i, s)] \leftarrow \langle v, a_p \rangle,$$

where *cache* is the cache table and $hash(i, s)$ gives the hash value of relevant variables of state s in subtask M_i. The next time the evaluation of state s under the same condition is requested, the cached result will be returned immediately with a probability

Table II. Empirical Results in the Taxi Domain

Algorithm	Trials	Average Rewards	Offline Time	Average Online Time
MAXQ-OP	1,000	3.93 ± 0.16	—	0.20 ± 0.16 ms
LRTDP	1,000	3.71 ± 0.15	—	64.88 ± 3.71 ms
AOT	1,000	3.80 ± 0.16	—	41.26 ± 2.37 ms
UCT	1,000	-23.10 ± 0.84	—	102.20 ± 4.24 ms
DNG-MCTS	1,000	-3.13 ± 0.29	—	213.85 ± 4.75 ms
R-MAXQ	100	3.25 ± 0.50	1200 ± 50 episodes	-
MAXQ-Q	100	0.0 ± 0.50	1, 600 episodes	-

Note: The optimal value of Average Rewards is 4.01 ± 0.15 averaged over 1,000 trials.

of 0.9. This strategy results in a huge number of search tree prunings. The key observation is that if a subtask has been completely evaluated before (i.e., evaluated with $d[i] = 0$), then it is most likely that we do not need to reevaluate it again in the near future.

In the experiments, we run several trials for each comparison algorithm with randomly selected initial states and report the average returns (accumulated rewards) and time usage over all trials in Table II. Offline time is the computation time used for offline algorithms to converge before evaluation online, and online time is the overall running time from initial state to terminal state for online algorithms when evaluating online. LRTDP [Bonet and Geffner 2003], AOT [Bonet and Geffner 2012], UCT [Kocsis and Szepesvári 2006], and DNG-MCTS [Bai et al. 2013a] are all trial-based anytime algorithms. The number of iterations for each action selection is set to 100. The maximal search depth is 100. They are implemented as online algorithms. A *min-min* heuristic [Bonet and Geffner 2003] is used to initialize new nodes in LRTDP and AOT, and a *min-min* heuristic–based greedy policy is used as the default rollout policy for UCT and DNG-MCTS. Note that both UCT and DNG-MCTS are Monte Carlo algorithms that only have knowledge of a generative model (a.k.a. a simulator) instead of the explicit transition model of the underlying MDP. R-MAXQ and MAXQ-Q are HRL algorithms. The results are taken from Jong and Stone [2008]. All experiments are run on a Linux 3.8 computer with 2.90GHz quad-core CPUs and 8GB RAM. It can be seen from the results that MAXQ-OP is able to find the near-optimal policy of the Taxi domain online with the value of 3.93 ± 0.16, which is very close to the optimal value of 4.01 ± 0.15. In particular, the time usage for MAXQ-OP is extremely less than other online algorithms compared in the experiments. These comparisons empirically confirm the effectiveness of MAXQ-OP in terms of its ability to exploit the hierarchical structure of the underlying problem while performing online decision making.

Furthermore, we have also introduced an extension of the Taxi domain to test our algorithm more thoroughly when scaling to increasingly complex problems. In the extended eTaxi[n] problem, the grid world size is $n \times n$. The four terminals R, G, Y, and B, are arranged at positions $(0, 0)$, $(0, n-1)$, $(n-2, 0)$, and $(n-1, n-1)$, respectively. There are three walls, each with length $\lfloor \frac{n-1}{2} \rfloor$ started at positions in between $(0, 0)$ and $(1, 0)$, $(1, n-1)$ and $(2, n-1)$, and $(n-3, 0)$ and $(n-2, 0)$, respectively. The action space remains the same as in the original Taxi domain. The transition and reward functions are extended accordingly such that if $n = 5$, eTaxi[n] reduces to the original Taxi problem. The same experiments with the *min-min* heuristic for all online algorithms are conducted over different sizes of eTaxi, ranging from $n = 5$ to 15. MAXQ-OP is also implemented with a *min-min* heuristic in this experiment, as the walls are relatively much longer in eTaxi with larger sizes, such that the simple Manhattan distance–based heuristic is not sufficient for MAXQ-OP. The average returns and online time usages are reported in Figure 3(a) and (b). It can be seen from the results that MAXQ-OP has

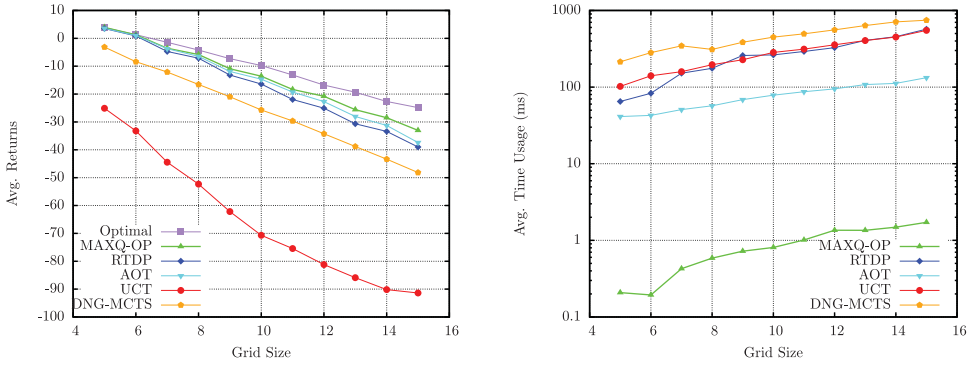


Fig. 3. Average returns (a) and average online time usages in eTaxi (b).

competitive performance in terms of average returns comparing to LRTDP and AOT, but with significantly less time usage. To conclude, MAXQ-OP is more time efficient due to the hierarchical structure used and the state abstraction and subtask sharing made in the algorithm.

7. CASE STUDY: ROBOCUP 2D

As one of the oldest leagues in RoboCup, the soccer simulation 2D league has achieved great successes and inspired many researchers all over the world to engage themselves in this game each year [Nardi and Iocchi 2006; Gabel and Riedmiller 2011]. Hundreds of research articles based on RoboCup 2D have been published.[3] Compared to other leagues in RoboCup, the key feature of RoboCup 2D is the abstraction made by the simulator, 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 planning, learning, and cooperation. For example, Stone et al. [2005] 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. They have also extended their work to a more general task named *half-field offense* [Kalyanakrishnan et al. 2007]. In the same reinforcement learning track, Riedmiller et al. [2009] have developed several effective techniques to learn mainly low-level skills in RoboCup 2D, such as intercepting and hassling.

In this section, we present our long-term effort of applying MAXQ-OP to the planning problem in RoboCup 2D. The MAXQ-OP–based overall decision framework has been implemented in our team WrightEagle, which has participated in annual RoboCup competitions since 1999, winning five world championships and named runner-up five times in the past 10 years.

To apply MAXQ-OP, we must first model the planning problem in RoboCup 2D as a MDP. This is nontrivial given the complexity of RoboCup 2D. We show how RoboCup 2D can be modeled as an MDP in Appendix A and what we have done in our team WrightEagle. Based on this, the following sections describe the successful application of MAXQ-OP in the RoboCup 2D domain.

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

7.1. Solution with MAXQ-OP

Here we describe our effort in applying MAXQ-OP to the RoboCup 2D domain in detail. First, a series of subtasks at different levels are defined as the building blocks for constructing the overall MAXQ hierarchy, listed as follows:

—kick, turn, dash, and tackle: These actions are the lowest-level primitive actions originally defined by the server. A local reward of –1 is assigned to each primitive action when performed to guarantee that the found online policy for high-level skills will try to reach respective (sub)goal states as fast as possible. kick and tackle ignore all the state variables except the state of the agent itself and the ball state; turn and dash only consider the state of the agent itself.
—KickTo, TackleTo, and NavTo: In the KickTo and TackleTo subtasks, the goals are to finally kick or tackle the ball in given direction with specified velocities. To achieve the goals, particularly in KickTo behavior, multiple steps of adjustment by executing turn or kick actions are usually necessary. The goal of the NavTo subtask (as shown in Figure 6(a)) is to move the agent from its current location to a target location as fast as possible by executing turn and dash actions under the consideration of action uncertainties. Subtasks KickTo and TackleTo terminate if the ball is no longer kickable/tackleable for the agent, and NavTo terminates if the agent has arrived the target location within a distance threshold. KickTo and TackleTo only consider the states of the agent itself and the ball; NavTo ignores all state variables except the state of the agent itself.
—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 (as shown in Figure 6(b)), (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 in formation when 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 keep an eye on close opponents, and (9) Formation is to maintain in formation when defending. Active states for Shoot, Dribble, and Pass are that the ball is kickable for the agent, whereas for other behaviors, the ball is not kickable for the agent. Shoot, Dribble, and Pass terminate when the ball is not kickable for the agent; Intercept terminates if the ball is kickable for the agent or is intercepted by any other players; Position terminates when the ball if kickable for the agent or is intercepted by any opponents; and other defending behaviors terminate when the ball is intercepted by any teammates (including the agent). These high-level behaviors will only consider relevant state variables—for example, Shoot, Dribble, and Intercept only consider the state of the agent, the state of the ball, and the states of other opponent players if they are close to the ball; Block, Trap, and Mark only consider the state of the agent itself and the state of one target opponent player; and Pass, Position, and Formation need to consider the states of all players and the ball.
—Attack and Defense: The goal of Attack is to attack opponents to finally score by planning on attacking behaviors, whereas the goal of Defense is to defend against opponents to prevent scoring of opponents by taking defending behaviors. Attack terminates if the ball is intercepted by any opponents, whereas Defense terminates if the ball is intercepted by any teammates (including the agent). All state variables are relevant to Attack and Defense, as they will be used by child actions.
—Root: This is the root task of the agent. A hand-coded strategy is used in Root task. It evaluates the Attack subtask first to see whether it is possible to attack; otherwise, it will select the Defense subtask. Roottask cannot ignore any state variables.

The task graph of the MAXQ hierarchical structure in the WrightEagle team is shown in Figure 4, where a parenthesis after a subtask's name indicates that the subtask takes

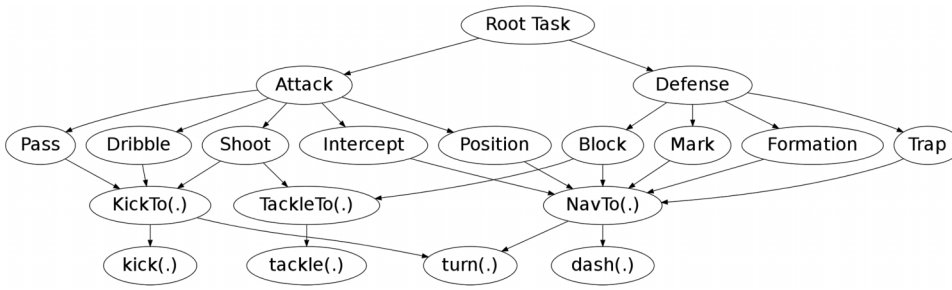


Fig. 4. MAXQ task graph for WrightEagle.

parameters. Take Attack, Pass, and Intercept as examples. For convenience, we assume that the agent always has an appropriate body angle when the ball is kickable for the agent, so the KickTo behavior only needs to plan kick actions. Let $\boldsymbol{s}$ be the estimated joint state; according to Equations (8), (9), and (13), we have

$$Q^*(\text{Root}, \boldsymbol{s}, \text{Attack}) = V^*(\text{Attack}, \boldsymbol{s}) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \text{Attack}) V^*(\text{Root}, \boldsymbol{s}'), \tag{21}$$

$$V^*(\text{Root}, \boldsymbol{s}) = \max\{Q^*(\text{Root}, \boldsymbol{s}, \text{Attack}), Q^*(\text{Root}, \boldsymbol{s}, \text{Defense})\}, \tag{22}$$

$$\begin{aligned} V^*(\text{Attack}, \boldsymbol{s}) = \max\{&Q^*(\text{Attack}, \boldsymbol{s}, \text{Pass}), Q^*(\text{Attack}, \boldsymbol{s}, \text{Dribble}), Q^*(\text{Attack}, \boldsymbol{s}, \text{Shoot}), \\ &Q^*(\text{Attack}, \boldsymbol{s}, \text{Intercept}), Q^*(\text{Attack}, \boldsymbol{s}, \text{Position})\}, \end{aligned} \tag{23}$$

$$Q^*(\text{Attack}, \boldsymbol{s}, \text{Pass}) = V^*(\text{Pass}, \boldsymbol{s}) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \text{Pass}) V^*(\text{Attack}, \boldsymbol{s}'), \tag{24}$$

$$Q^*(\text{Attack}, \boldsymbol{s}, \text{Intercept}) = V^*(\text{Intercept}, \boldsymbol{s}) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \text{Intercept}) V^*(\text{Attack}, \boldsymbol{s}'), \tag{25}$$

$$V^*(\text{Pass}, \boldsymbol{s}) = \max_{\text{position } p} Q^*(\text{Pass}, \boldsymbol{s}, \text{KickTo}(p)), \tag{26}$$

$$V^*(\text{Intercept}, \boldsymbol{s}) = \max_{\text{position } p} Q^*(\text{Intercept}, \boldsymbol{s}, \text{NavTo}(p)), \tag{27}$$

$$Q^*(\text{Pass}, \boldsymbol{s}, \text{KickTo}(p)) = V^*(\text{KickTo}(p), \boldsymbol{s}) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \text{KickTo}(p)) V^*(\text{Pass}, \boldsymbol{s}'), \tag{28}$$

$$Q^*(\text{Intercept}, \boldsymbol{s}, \text{NavTo}(p)) = V^*(\text{NavTo}(p), \boldsymbol{s}) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \text{NavTo}(p)) V^*(\text{Intercept}, \boldsymbol{s}'), \tag{29}$$

$$V^*(\text{KickTo}(p), \boldsymbol{s}) = \max_{\text{power } a,\ \text{angle } \theta} Q^*(\text{KickTo}(p), \boldsymbol{s}, \text{kick}(a, \theta)), \tag{30}$$

$$V^*(\text{NavTo}(p), \boldsymbol{s}) = \max_{\text{power } a,\ \text{angle } \theta} Q^*(\text{NavTo}(p), \boldsymbol{s}, \text{dash}(a, \theta)), \tag{31}$$

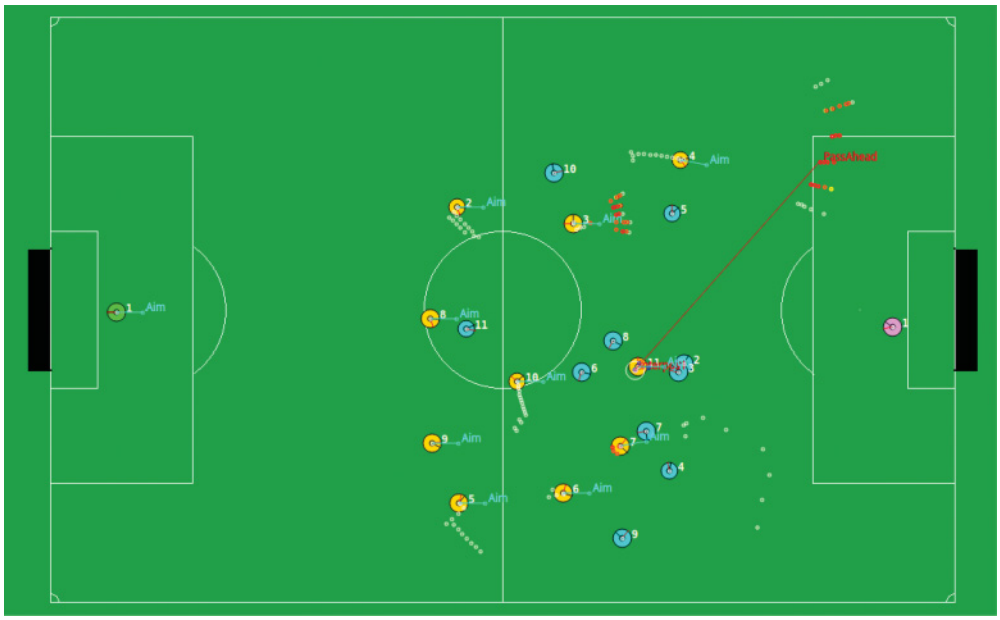


Fig. 5. Hierarchical planning in pass behavior. ©The RoboCup Federation.

$$Q^*(\mathsf{KickTo}(p), \boldsymbol{s}, \mathsf{kick}(a, \theta)) = R(s, \mathsf{kick}(a, \theta)) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \mathsf{kick}(a, \theta)) V^*(\mathsf{KickTo}(p), \boldsymbol{s}'), \tag{32}$$

$$Q^*(\mathsf{NavTo}(p), \boldsymbol{s}, \mathsf{dash}(a, \theta)) = R(s, \mathsf{dash}(a, \theta)) + \sum_{s'} P_t(s' \mid \boldsymbol{s}, \mathsf{dash}(a, \theta)) V^*(\mathsf{NavTo}(p), \boldsymbol{s}'). \tag{33}$$

As an example, Figure 5 shows the hierarchical planning process in Pass behavior. When player 11 is planning the Pass behavior, the agent will evaluate the possibility of passing the ball to each teammate; for each teammate, the agent will propose a set of pass targets to kick the ball; and for each target, the agent will plan a sequence of kick actions to kick the ball to that position as fast as possible in the KickTo subtask. The set of targets proposed for each teammate is generated by using a hill-climbing method, which tries to find a most valuable target for a particular teammate in terms of an evaluation function defined by recursive value functions of low-level subtasks and the completion function of the Pass behavior, which strongly depends on the probability of success for the passing target.

As mentioned, the local reward for kick action is $R(s, \mathsf{kick}(a, \theta)) = -1$, and the respective termination distribution $P_t(s' \mid \boldsymbol{s}, \mathsf{kick}(a, \theta))$ is totally defined by the server. Subtask $\mathsf{KickTo}(p)$ successfully terminates if the ball after a kick is moving approximately toward position p. Thus, Equation (30) gives the negative number of cycles needed to kick the ball to position p. Subtask Pass terminates if the ball is not kickable for the agent, and the control returns to Attack, which will then evaluate whether the agent should do Intercept in case the ball is slipped from the agent or Position to keep in attacking formation. Subtask $\mathsf{NavTo}(p)$ terminates if the agent is almost at position p. Similarly, we have $R(s, \mathsf{dash}(a, \theta)) = -1$, and $P_t(s' \mid s, \mathsf{dash}(a, \theta))$ is defined by the

server. Equation (31) gives the negative number of cycles needed to move the agent to position p from its current position in the joint state $\boldsymbol{s}$. Equation (27) gives the negative value of the expected number of cycles needed to intercept the ball. The Attack behavior terminates if the ball is intercepted by the opponent team. When terminating, the control returns to the Root behavior, which will consider taking defending behaviors by planning in Defense task. The Defense behavior terminates if the ball is intercepted by the agent or any teammates.

To approximate termination distributions online for behaviors, a fundamental probability that needs to be estimated is the probability that a moving ball will be intercepted by a player p (either a teammate or an opponent). Let $b = (b_x, b_y, b_{\dot{x}}, b_{\dot{y}})$ be the state of the ball and $p = (p_x, p_y, p_{\dot{x}}, p_{\dot{y}}, p_\alpha, p_\beta)$ be the state of the player. Let $\Pr(p \leftarrow b \mid b, p)$ denote the probability that the ball will be intercepted by player p. Formally, $\Pr(p \leftarrow b \mid b, p) = \max\{\Pr(p \leftarrow b, t \mid b, p)\}$, where $\Pr(p \leftarrow b, t \mid b, p)$ is the probability that player p will intercept the ball at cycle t from now, which is approximated as $\Pr(p \leftarrow b, t \mid b, p) = g(t - f(p, b_t))$, where b_t is the ball's predicted state in cycle t, $f(p, b_t)$ returns the estimated number of cycles needed for the player moving at its maximal speed from the current position (p_x, p_y) to the ball's position in cycle t, and $g(\delta)$ gives the estimated probability given that the cycle difference is δ. The intercepting probability function $g(\delta)$ is illustrated in Figure 7. Given the intercepting probabilities, we approximate termination distributions for other behaviors, for example, as

$$P_t(\boldsymbol{s}' \mid \boldsymbol{s}, \text{Attack}) = 1 - \prod_{\text{opponent } o} (1 - \Pr(o \leftarrow b \mid b, o)), \tag{34}$$

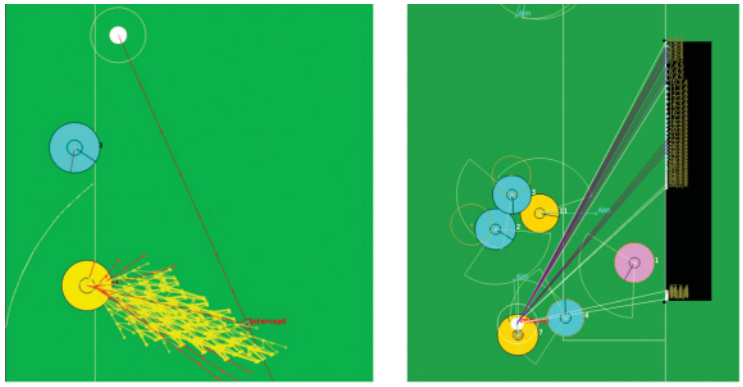
$$P_t(s' \mid \boldsymbol{s}, \text{Defense}) = 1 - \prod_{\text{teammate } t} (1 - \Pr(t \leftarrow b \mid b, t)), \tag{35}$$

$$P_t(s' \mid \boldsymbol{s}, \text{Intercept}) = \mathbf{1}[\exists \text{player } i : i \leftarrow b] P_t(i \leftarrow b \mid b, i) \prod_{\text{player } p \neq i} (1 - \Pr(p \leftarrow b \mid b, p)), \tag{36}$$

$$P_t(s' \mid \boldsymbol{s}, \text{Position}) = \mathbf{1}[\exists \text{non-teammate } i : i \leftarrow b] \Pr(i \leftarrow b \mid b, i) \prod_{\text{player } p \neq i} \times (1 - \Pr(p \leftarrow b \mid b, p)), \tag{37}$$

where $b = \boldsymbol{s}[0]$ is the ball state. Some other probabilities, such as the probability that the moving ball will finally go through the opponent goal, are approximated offline, taking advantage of some statistical methods.

State abstractions are implicitly introduced by the task hierarchy. For example, only the agent's self-state and the ball's state are relevant when evaluating Equations (30) and (27). When enumerating power and angle for the kick and dash actions, only a set of discretized parameters is considered. This leads to limited precision of the solution, yet is necessary to deal with continuous action space and meet real-time constraints. To deal with the large action space, heuristic methods are critical to apply 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 (as shown in Figure 5), and A* search is used to search over the action space of dash and turn for the NavTo subtask in the discretized state space. A search tree of the NavTo subtask is shown in Figure 6(a), where yellow lines represent the state transitions in the search tree. For Shoot behavior, it turns out the we only need to evaluate a set of dominant positions in terms of the joint probability that the ball



(a) A* search tree in NavTo (b) Prunning method in Shoot

Fig. 6. Heuristic search in action spaces. ©The RoboCup Federation.

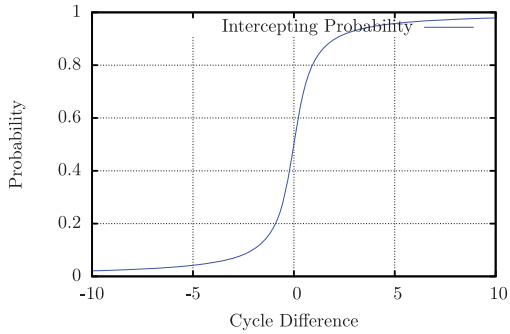


Fig. 7. Intercepting probability estimation.

can finally go through the opponent goal area, without being touched by any of the opponent players, including the goalie, which is depicted in Figure 6(b), where only a small set of positions linked with purple lines is evaluated.

Another important component of applying MAXQ-OP is to estimate value functions for subtasks using heuristics when the search depth exceeds the maximal depth allowed. Taking the Attack task as an example, we introduce *impelling speed* to estimate $V^*(\text{Attack}, \boldsymbol{s}_t)$, where $\boldsymbol{s}_t$ is the state to be evaluated in t cycles from now. Given current state $\boldsymbol{s}$ and the state $\boldsymbol{s}'$ to be evaluated, impelling speed is formally defined as

$$impelling\_speed(\boldsymbol{s}, \boldsymbol{s}', \alpha) = \frac{dist(\boldsymbol{s}, \boldsymbol{s}', \alpha) + pre\_dist(\boldsymbol{s}', \alpha)}{step(\boldsymbol{s}, \boldsymbol{s}') + pre\_step(\boldsymbol{s}')}, \tag{38}$$

where α is a global attacking direction (named *aim-angle* in our team), $dist(\boldsymbol{s}, \boldsymbol{s}', \alpha)$ is the ball's running distance projected in direction α from state $\boldsymbol{s}$ to state $\boldsymbol{s}'$, $step(\boldsymbol{s}, \boldsymbol{s}')$ is the running steps from state $\boldsymbol{s}$ to state $\boldsymbol{s}'$, $pre\_dist(\boldsymbol{s}')$ estimates remaining distance projected in direction α from state $\boldsymbol{s}'$ that the ball can be impelled without being intercepted by opponents, and $pre\_step(\boldsymbol{s}')$ is the respective remaining steps. The aim-angle α in state $\boldsymbol{s}$ is determined by an $aim\_angle(\boldsymbol{s})$ function. $V^*(\text{Attack}, \boldsymbol{s})$ is then approximated as

$$V^*(\text{Attack}, \boldsymbol{s}_t) = impelling\_speed(\boldsymbol{s}_0, \boldsymbol{s}_t, aim\_angle(\boldsymbol{s}_0)), \tag{39}$$

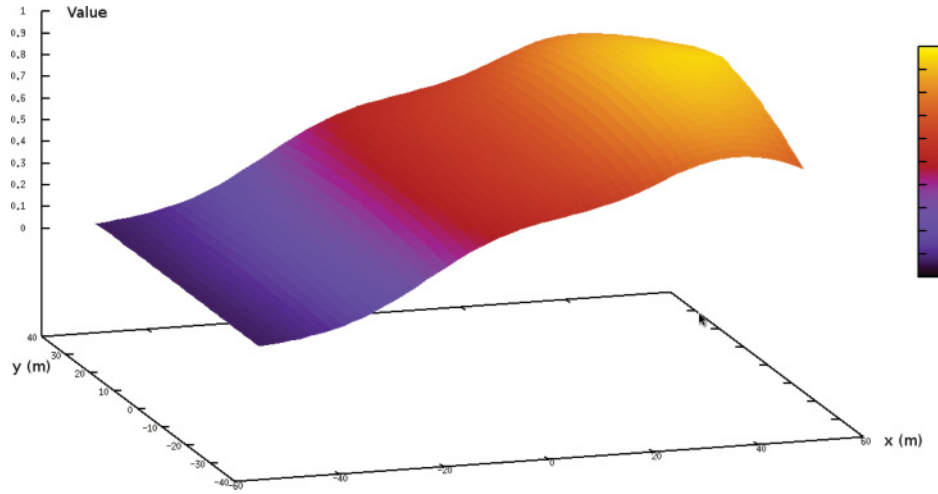


Fig. 8. A heuristic function used in defending behaviors.

where s_0 is the current state—that is, the root of the search tree. The value of $impelling\_speed(\boldsymbol{s}_0, \boldsymbol{s}_t, aim\_angle(\boldsymbol{s}_0))$ implies the fact that the faster the ball is moving in the attacking direction, the more attacking opportunities there could be, and the more valuable the state $\boldsymbol{s}_t$ is.

For defending behaviors, a value function over ball positions is used as the heuristic function, which is shown in Figure 8. The figure reflects the fact that the positions in opponent goal area are the most valuable positions, whereas the positions around our bottom line are very dangerous when defending.

7.2. Empirical Evaluation

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

—FULL: This is exactly the full version of our team, where a MAXQ-OP–based online planning framework is implemented as the key component. The Attack behavior chooses among attacking behaviors such as Shoot, Pass, and Dribble according to their returned values in the MAXQ-OP framework.
—RANDOM: This is almost 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 Pass or Dribble with uniform probabilities.
—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.

Notice that 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 made based on the values returned from lower-level subtasks (i.e., the solutions found by `EvaluateStateInSubtask(`Pass, $\cdot$, $\cdot$`)` and `EvaluateStateInSubtask(` Dribble, $\cdot$, $\cdot$ `)` in the MAXQ-OP framework). Although RANDOM and HAND-CODED have different Pass-Dribble selection strategies, the remaining subtasks, including Shoot, Pass, Dribble, and Intercept and all defending behaviors, remain the same as in the FULL version.

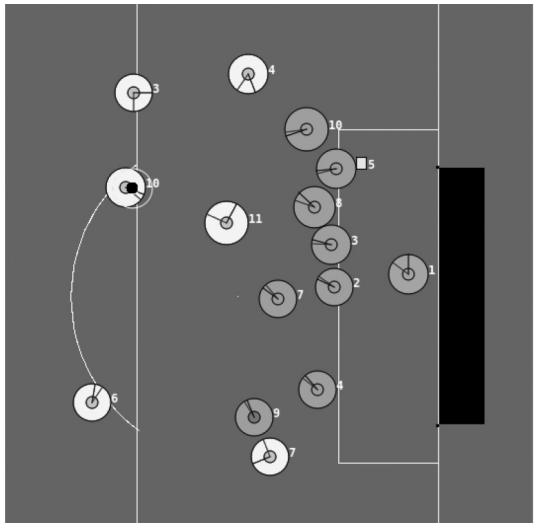


Fig. 9. A selected initial scenario from RoboCup 2011. ©The RoboCup Federation.

Table III. Empirical Results of WrightEagle in the Scenario Test

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

For each version of the testing team, we use an offline coach (also known as a trainer in RoboCup 2D) to independently run the team against the Helios11 binary (which has participated in RoboCup 2011 and won second place) for 100 episodes. Each episode begins with a fixed scenario given by a complete joint state taken from the final match of RoboCup 2011 and ends when (1) our team scores a goal, denoted by **success**; (2) the ball's x coordination is smaller than –10.0, denoted by **failure**; or (3) the episode runs longer than 200 cycles, denoted by **timeout**. Note that although all of the episodes begin with the same scenario, none of them is identical due to the uncertainties of the environment.

The initial state in the selected scenario, which is at cycle #3142 of that match, is shown in Figure 9, in which white circles represent our players, gray circles represent opponents, and the small black circle represents the ball. At this cycle, our player 10 is holding the ball, whereas 9 opponents (including the goalie) are blocking in front of their goal area. In RoboCup 2011, teammate 10 passed the ball directly to teammate 11. When teammate 11 had the ball, it passed the ball back to teammate 10 after dribbling for a number of cycles. When teammate 11 moved to an appropriate position, teammate 10 passed the ball again to teammate 11. Finally, teammate 11 executed a Tackle action to shoot at cycle #3158 and successfully scored five cycles later.

The experimental results are presented in Table III, from which we can see that the FULL version of our team outperforms both RANDOM and HAND-CODED with an increase of the chance of **success** by 86.7% and 64.7%, respectively. The performances of RANDOM and HAND-CODED are actually very close, the reason being that they are sharing the same task hierarchy and all of the same subtasks as in the Full version except the Pass-Dribble selection strategy. We find that the local selection strategy between Pass and Dribble plays a key role in the decision of Attack and affects the

Table IV. Empirical Results of WrightEagle in the Full Game Test

Opponent Team	Games	Average Goals	Average Points	Winning Rate
Brainstormers08	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	$91.0 \pm 5.6\%$

Table V. Historical Results of WrightEagle in RoboCup Annual Competitions Since 2005

Competitions	Games	Points	Goals	Win	Draw	Lost	Average Points	Average Goals
RoboCup 2005	19	47	84 : 16	15	2	2	2.47	4.42 : 0.84
RoboCup 2006	14	38	57 : 6	12	2	0	2.71	4.07 : 0.43
RoboCup 2007	14	34	125 : 9	11	1	2	2.42	8.92 : 0.64
RoboCup 2008	16	40	74 : 18	13	1	2	2.50	4.63 : 1.13
RoboCup 2009	14	36	81 : 17	12	0	2	2.57	5.79 : 1.21
RoboCup 2010	13	33	123 : 7	11	0	2	2.54	9.47 : 0.54
RoboCup 2011	12	36	151 : 3	12	0	0	3.00	12.6 : 0.25
RoboCup 2012	21	58	104 : 18	19	1	1	2.76	4.95 : 0.86
RoboCup 2013	19	53	104 : 9	17	2	0	2.79	5.47 : 0.47

final performance substantially. The results indicate that the MAXQ-OP–based local selection strategy between Pass and Dribble is sufficient for the Attack behavior to achieve high performance. Recursively, this is also true for other subtasks over the resulting task hierarchy, such as Defense, Shoot, and Pass. The comparison with the HAND-CODED strategy also indicates that the MAXQ-OP algorithm gets leverage not only from the hierarchical structure but also from the algorithm itself in terms of the MAXQ-OP–derived action-selection strategy. To conclude, MAXQ-OP is able to be the key to success of our team in this scenario test.

We have also tested the FULL version of our team in full games against four high-quality RoboCup 2D teams, namely Brainstormers08, Helios10, Helios11, and Oxsy11. Brainstormers08 and Helios10 were the champions of RoboCup 2008 and RoboCup 2010, respectively. In the experiments, we independently ran our team against one of the official binaries for 100 games under the same hardware conditions. Table IV summarizes the detailed empirical result. The winning rate is defined as $p = n/N$, where n is the number of games that we have won and N is the total number of games. It can be seen from the result that our team substantially outperforms other teams in terms of the winning rate. Specifically, our team has about 82.0%, 93.0%, 83.0%, and 91.0% of the chances to win over BrainsStomers08, Helios10, Helios11 and Oxsy11, respectively. Table V reports the historical results of WrightEagle in RoboCup 2D annual competitions since 2005. It can be seen from the result that our team has reached outstanding performance in RoboCup competitions: we rarely lose or draw in the competitions.

Although 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 hierarchical structure we used and the abstraction we made for the actions and states. The key advantage of applying MAXQ-OP in RoboCup 2D is to provide a principled framework for conducting the online search process over a task hierarchy. Therefore, the team can search for a strategy-level solution automatically online by being given the predefined task hierarchy. To the best of our knowledge, most of the current RoboCup teams develop their agents based on hand-coded rules. Overall, the goal of this case study is twofold: (1) it demonstrates the scalability and efficiency of MAXQ-OP for solving a large real-world application such as RoboCup 2D, and (2) it presents a

decision-theoretic solution for developing a RoboCup soccer team, which is general for programming high-level strategies.

8. CONCLUSIONS

This article presents MAXQ-OP—a novel online planning algorithm that benefits from the advantage of hierarchical decomposition. It recursively expands the search tree online by following the underlying MAXQ task hierarchy. This is efficient, as only relevant states and actions are considered according to the task hierarchy. Another contribution of this work is the completion function approximation method, which makes it possible to apply MAXQ-OP online. The key observation is that the termination distribution is relatively easy to be approximated either online or offline given domain knowledge. The empirical results show that MAXQ-OP is able to find a near-optimal policy online for the Taxi domain and reaches outstanding performance in the highly complex RoboCup 2D domain. The experimental results confirm the soundness and stability of MAXQ-OP to solve large MDPs by utilizing hierarchical structure. In future work, we plan to theoretically analyze MAXQ-OP with different task priors and test them on more real-world applications.

ACKNOWLEDGMENTS

The authors thank Changjie Fan, Ke Shi, Haochong Zhang, Guanghui Lu, Rongya Chen, Xiao Li, and other members for their contributions to the WrightEagle team. The authors would like to thank Manuela Veloso, Shlomo Zilberstein, Peter Stone, and the CORAL research group at CMU for the helpful discussions with them. The authors also want to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

David Andre and Stuart J. Russell. 2002. *State Abstraction for Programmable Reinforcement Learning Agents*. Technical Report. University of California at Berkeley.

Mehran Asadi and Manfred Huber. 2004. State space reduction for hierarchical reinforcement learning. In *Proceedings of the FLAIRS Conference*. 509–514.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47, 2, 235–256.

Aijun Bai, Feng Wu, and Xiaoping Chen. 2012. Online planning for large MDPs with MAXQ decomposition (extended abstract). In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'12)*. 1215–1216.

Aijun Bai, Feng Wu, and Xiaoping Chen. 2013a. Bayesian mixture modelling and inference based Thompson sampling in Monte-Carlo tree search. In *Advances in Neural Information Processing Systems 26*. 1646–1654.

Aijun Bai, Feng Wu, and Xiaoping Chen. 2013b. Towards a principled solution to simulated robot soccer. In *RoboCup 2012: Robot Soccer World Cup XVI*. Lecture Notes in Computer Science, Vol. 7500. Springer, 141–153.

Bram Bakker and Jürgen Schmidhuber. 2004. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems*. 438–445.

Bram Bakker, Zoran Zivkovic, and Ben Krose. 2005. Hierarchical dynamic programming for robot path planning. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*. IEEE, Los Alamitos, CA, 2756–2761.

Jennifer Barry. 2009. *Fast Approximate Hierarchical Solution of MDPs*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

Jennifer Barry, Leslie Kaelbling, and Tomas Lozano-Perez. 2011. DetH*: Approximate hierarchical solution of large Markov decision processes. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 1928–1935.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72, 1–2, 81–138.

Andrew G. Barto and Sridhar Mahadevan. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems* 13, 4, 341–379.

Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Dimitri P. Bertsekas. 1996. *Dynamic Programming and Optimal Control*. Athena Scientific.

Blai Bonet and Hector Geffner. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*.

Blai Bonet and Hector Geffner. 2012. Action selection for MDPs: Anytime AO* vs. UCT. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1749–1755.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1, 1–43.

Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. 2001. Monte Carlo localization for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 2. IEEE, Los Alamitos, CA, 1322–1328.

Thomas G. Dietterich. 1999a. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Machine Learning Research* 13, 1, 63.

Thomas G. Dietterich. 1999b. State abstraction in MAXQ hierarchical reinforcement learning. arXiv preprint cs/9905015.

Carlos Diuk, Alexander L. Strehl, and Michael L. Littman. 2006. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM, New York, NY, 313–319.

Zohar Feldman and Carmel Domshlak. 2012. Simple regret optimization in online planning for Markov decision processes. arXiv preprint 1206.3382.

Zhengzhu Feng and Eric A. Hansen. 2002. Symbolic heuristic search for factored Markov decision processes. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02)*. 455–460.

Thomas Gabel and Martin Riedmiller. 2011. On progress in RoboCup: The simulation league showcase. In *RoboCup 2010: Robot Soccer World Cup XIV*. Lecture Notes in Computer Science, Vol. 6556. Springer, 36–47.

Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11, 1856–1875.

Eric A. Hansen and Shlomo Zilberstein. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129, 1–2, 35–62.

Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. 1998. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*. 220–229.

Bernhard Hengst. 2002. Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the 19th International Conference on Machine Learning (ICML'02)*, Vol. 2. 243–250.

Bernhard Hengst. 2004. Model approximation for HEXQ hierarchical reinforcement learning. In *Machine Learning: ECML 2004*. Lecture Notes in Computer Science, Vol. 3201. Springer, 144–155.

Bernhard Hengst. 2007. Safe state abstraction and reusable continuing subtasks in hierarchical reinforcement learning. In *AI 2007: Advances in Artificial Intelligence*. Lecture Notes in Computer Science, Vol. 4830. Springer, 58–67.

Nicholas K. Jong and Peter Stone. 2008. Hierarchical model-based reinforcement learning: R-max + MAXQ. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, New York, NY, 432–439.

Anders Jonsson and Andrew Barto. 2006. Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research* 7, 2259–2301.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101, 1–2, 99–134.

Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. 2007. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In *RoboCup 2006: Robot Soccer World Cup X*. Lecture Notes in Computer Science, Vol. 4434. Springer, 72–85.

Michael Kearns, Yishay Mansour, and Andrew Y. Ng. 1999. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, Vol. 2. 1324–1331.

Thomas Keller and Malte Helmert. 2013. Trial-based heuristic tree search for finite horizon MDPs. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*. 135–143.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*. 282–293.

Lihong Li, Thomas J. Walsh, and Michael L. Littman. 2006. Towards a unified theory of state abstraction for MDPs. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics (ISAIM'06)*.

Michael L. Littman, Thomas L. Dean, and Leslie P. Kaelbling. 1995. On the complexity of solving Markov decision problems. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*. 394–402.

Victoria Manfredi and Sridhar Mahadevan. 2005. Hierarchical reinforcement learning using graphical models. In *Proceedings of the ICML 2005 Workshop on Rich Representations for Reinforcement Learning*. 39–44.

H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. 2005. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd International Conference on Machine Learning*. ACM, New York, NY, 569–576.

Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. 2008. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, New York, NY, 648–655.

Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. 2011. Automatic discovery and transfer of task hierarchies in reinforcement learning. *AI Magazine* 32, 1, 35.

Daniele Nardi and Luca Iocchi. 2006. Artificial intelligence in RoboCup. In *Reasoning, Action and Interaction in AI Theories and Systems*. Lecture Notes in Computer Science, Vol. 4155. Springer, 193–211.

Nils J. Nilsson. 1982. *Principles of Artificial Intelligence*. Springer.

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. 2009. Reinforcement learning for robot soccer. *Autonomous Robots* 27, 1, 55–73.

Scott Sanner, Robby Goetschalckx, Kurt Driessens, and Guy Shani. 2009. Bayesian real-time dynamic programming. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*. 1784–1789.

Özgür Şimşek, Alicia P. Wolfe, and Andrew G. Barto. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*. ACM, New York, NY, 816–823.

Martin Stolle. 2004. *Automated Discovery of Options in Reinforcement Learning*. Ph.D. Dissertation. McGill University.

Peter Stone. 2000. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.

Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. 2005. Reinforcement learning for RoboCup soccer keepaway. *Adaptive Behavior* 13, 3, 165–188.

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*, Vol. 116. Cambridge University Press.

Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112, 1, 181–211.

Matthew E. Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, 1633–1685.

Received April 2014; revised October 2014; accepted January 2015