Online Planning for Large MDPs with MAXQ Decomposition

Aijun Bai, Feng Wu, and Xiaoping Chen

Multi-Agent Systems Lab, Department of Computer Science,
University of Science and Technology of China
Hefei, 230026, China
baj@mail.ustc.edu.cn, xpchen@ustc.edu.cn

Abstract. Markov decision processes (MDPs) provide an expressive framework for planning in stochastic domains. However, exactly solving a large MDP is often intractable due to the curse of dimensionality. Online algorithms help overcome the high computational complexity by avoiding computing a policy for each possible state. Hierarchical decomposition is another promising way to help scale MDP algorithms up to large domains by exploiting their underlying structure. In this paper, we present an effort on combining the benefits of a general hierarchical structure based on MAXQ value function decomposition with the power of heuristic and approximate techniques for developing an online planning framework, called MAXQ-OP. The proposed framework provides a principled approach for programming autonomous agents in a large stochastic domain. We empirically evaluated our algorithm on the Taxi problem–a common benchmark for MAXQ–to show the efficiency of MAXQ-OP. We have also been conducting a long-term case-study with the RoboCup soccer simulation 2D domain, which is extremely larger than domains usually studied in the literature, as the major benchmark to this research. The case-study showed that the agents developed with this framework and the related techniques reached outstanding performances, showing its high scalability to very large domains.

Keywords: MDP, Online Planning, MAXQ, RoboCup 2D

1 Introduction

Markov decision processes (MDPs) have been proved to be a useful model for planning under uncertainty. Most of the existing approaches, such as linear programming, value iteration, and policy iteration [16], solve MDP problems *offline*. All these algorithms have to find a policy for the entire state space before actually interacting with the environments. When applying to large domains, exactly solving MDPs offline is often intractable due to the *curse of dimensionality*, i.e. the size of state space grows exponentially with the number of state variables. The very high computational complexity–typically polynomial

in the size of the state space [14]–may make it unacceptable in practice. Furthermore, small changes in the system's dynamics require recomputing the full policy, which makes it inapplicable to those domains where their dynamics may change constantly.

In contrast, *online* approaches try to alleviate this difficulty by focusing on computing a policy only for the current state. The key observation is that an agent could merely encounter a fraction of the overall states during execution. When interacting with the environment, online algorithms simultaneously evaluate all possible actions for the current state and select the "best" one. It recursively perform forward search on the reachable states by evaluating and updating the policy value in real-time. As shown in RTDP [6], LAO* [9] and UCT [13], heuristic techniques can be utilized in the search process to reduce time and memory usage. Online planning algorithms also have an advantage that explicit representations of policies are not necessary for them, so they can easily handle a continuous state or action space without discretization. Moreover, online algorithms can easily handle unpredicted dynamics changes, which makes them a preferable choice in many real-world applications.

Hierarchical decomposition is another well-known method for scaling MDP algorithms to very large problems. By given a hierarchical structure of the domain, it decomposes the original model into a set of subproblems (or subtasks) that can be more easily solved [4]. This method can benefit from the advantages of several useful techniques including *temporal abstraction*, *state abstraction* and *subtask sharing* [8]. In temporal abstraction, temporally-extended actions (also known as *options* or *macro-actions*), which may take numbers of steps to execute, are treated as primitive actions by the higher level of subtasks in the hierarchy. State abstraction aggregates the system states into *macro-states* by eliminating irrelevant state variables for subtasks. Subtask sharing allows the computed policy of one subtask to be reused by some other higher level tasks.

In this paper, we present a novel approach called MAXQ-OP for planning in stochastic domains modeled as MDPs. It combines the main advantages of online planning and hierarchical decomposition, namely MAXQ, to solve large MDPs. Unlike most of the existing MDP algorithms, MAXQ-OP runs the planning procedure online to find the best policy merely for the current step. In MAXQ-OP, only small parts of the policy space starting from the current state are visited. This is more efficient than the offline solutions, especially for large problems where computing the complete policy for every state is intractable. Another strength of MAXQ-OP is the ability to utilize hierarchical structures of the problems. In many real-world applications, such a hierarchical structure exists and can be used to speed up the planning. The key contribution of this paper lies in the overall framework for exploiting the hierarchical structure online and the approximation made for computing the *completion function*. We empirically evaluated our algorithm on the Taxi problem–a common benchmark for MAXQ–to show the efficiency of MAXQ-OP. We also performed a case-study on a very large domain–RoboCup soccer simulation 2D–to test the scalability of

MAXQ-OP. The empirical results confirmed the advantage of MAXQ-OP over existing approaches in the literature.

The remainder of this paper is organized as follows. Section 2 introduces background knowledge on Markov decision processes and MAXQ hierarchical decomposition. Section 3 describes our MAXQ-OP solution in details. Section 4 show the empirical results on Taxi domain, and Section 5 presents a case-study in RoboCup soccer simulation 2D domain. Finally, in Section 6, the paper is concluded with some discussion of the future work.

2 Background

In this section, we briefly review the MDP model [16] and the MAXQ hierarchical decomposition method [8].

2.1 MDP Framework

Formally, an MDP is defined as a tuple $\langle S, A, P, R \rangle$, where:

- S is a set of possible states of the environment.
- A is a set of actions to be performed by the agent.
- $P : S \times A \times S \to [0, 1]$ is the transition function with $P(s'|s, a)$ denoting the probability of transition to state s' after performing action a at state s.
- $R : S \times A \to \mathbb{R}$ is the reward function with $R(s, a)$ denoting the immediate reward that the agent would received after performing action a at state s.

A *policy* is a mapping from states to actions $\pi : S \to A$, with $\pi(s)$ denoting the action to be taken at state s. The *value function* $V^\pi(s)$ of a policy π is defined as the expected cumulative reward received by following π:

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

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

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s'). \tag{1}$$

Solving an MDP is equivalent to looking for an *optimal* policy π^* such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and any policy π. The optimal value function, denoted by $V^*(s)$, satisfes the following Bellman equation:

$$V^*(s) = \max_{a \in A} Q^*(s, a). \tag{2}$$

Then the optimal policy π^* is, for all $s \in S$:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s, a). \tag{3}$$

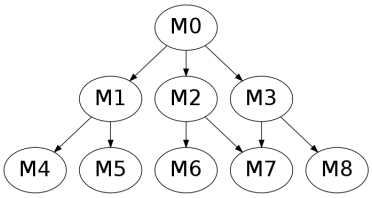

Fig. 1. An example of MAXQ task graph

In many real-world applications, there exists a pre-specified set of goal states, $G \subset S$. These states are typically *absorbing* states which: for every $g \in G$ and $a \in A$, $P(g, a, g) = 1$ and $R(g, a) = 0$. In this paper, we concentrate on undiscounted ($\gamma = 1$) goal-directed MDPs (also sometimes referred as *stochastic shortest path problems* [5]). It is shown that any MDP can be transformed into an equivalent undiscounted negative goal-directed MDP where the reward for non-goal states is strictly negative [2]. So undiscounted goal-directed MDP is actually a general formulation.

2.2 MAXQ Hierarchical Decomposition

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

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

It is worth pointing out that if a subtask has task parameters, then different binding of the parameters, may specify different instances of a subtask. Primitive actions are treated as primitive subtasks such that they are always executable, and will terminate immediately after execution. For primitive subtasks, the pseudo-reward function is uniformly zero. This hierarchical structure can be represented as a directed acyclic graph called the *task graph*. An example is shown in Figure 1, where M_0 is the root task with three children M_1, M_2, and M_3, which are subtasks sharing the lower-level primitive subtasks M_i ($4 \leq i \leq 8$) as their actions.

Given the hierarchical structure, a *hierarchical policy* π is defined as a set of policies for each subtask $\pi = \{\pi_0, \pi_1, \cdots, \pi_n\}$, where π_i is a mapping from active states to actions $\pi_i : S_i \to A_i$. The *projected value function* of policy π for subtask M_i in state s, $V^\pi(i, s)$, is defined as the expected value after following

policy π at state s until the subtask M_i terminates at one of its terminal states in G_i. Similarly, $Q^\pi(i, s, a)$ is the expected value by firstly performing action M_a at state s, and then following policy π until the termination of M_i. It is worth noting that $V^\pi(a, s) = R(s, a)$ if M_a is a primitive action $a \in A$.

Dietterich [8] has shown that the value function of policy π can be expressed recursively as:

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a) \tag{4}$$

where

$$V^\pi(i, s) = \begin{cases} R(s, i) & \text{if } M_i \text{ is primitive} \\ Q^\pi(i, s, \pi(s)) & \text{otherwise} \end{cases} \tag{5}$$

$C^\pi(i, s, a)$ is the *completion function* that estimates the cumulative reward received with the execution of action M_a before completing the subtask M_i, as defined below:

$$C^\pi(i, s, a) = \sum_{s', N} \gamma^N P(s', N|s, a) V^\pi(i, s'), \tag{6}$$

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

A *recursively optimal policy* π^* can be found by recursively computing the optimal projected value function as:

$$Q^*(i, s, a) = V^*(a, s) + C^{\pi^*}(i, s, a), \tag{7}$$

where

$$V^*(i, s) = \begin{cases} R(s, i) & \text{if } M_i \text{ is primitive} \\ \max_{a \in A_i} Q^*(i, s, a) & \text{otherwise} \end{cases}. \tag{8}$$

Then π_i^* for subtask M_i can be given by, for all $s \in S_i$:

$$\pi_i^*(s) = \underset{a \in A_i}{\operatorname{argmax}} Q^*(i, s, a). \tag{9}$$

3 Online Planning with MAXQ

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

3.1 Overview of MAXQ-OP

In general, online planning interleaves planning with execution and chooses the best action for the current step. Given the MAXQ hierarchy of an MDP, $M = \{M_0, M_1, \cdots, M_n\}$, the main procedure of MAXQ-OP evaluates each subtask by forward search to compute the recursive value functions $V^*(i, s)$ and $Q^*(i, s, a)$ online. This involves a complete search of all paths through the MAXQ hierarchy starting from the root task M_0 and ending with some primitive subtasks at the leaf nodes. After the search process, the best action $a \in A_0$ is selected for the root

Algorithm 1: `OnlinePlanning()`

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

$r \leftarrow 0$;
$s \leftarrow$ `GetInitState()`;
while $s \notin G_0$ **do**
  $\langle v, a_p \rangle \leftarrow$ `EvaluateState`$(0, s, [0, 0, \cdots, 0])$;
  $r \leftarrow r +$ `ExecuteAction`(a_p, s);
  $s \leftarrow$ `GetNextState()`;
return r;

Algorithm 2: `EvaluateState`(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** $\langle R(s, M_i), M_i \rangle$;
else if $s \notin S_i$ **and** $s \notin G_i$ **then return** $\langle -\infty, nil \rangle$;
else if $s \in G_i$ **then return** $\langle 0, nil \rangle$;
else if $d[i] \geq D[i]$ **then return** $\langle$`HeuristicValue`$(i, s), nil \rangle$;
else
  $\langle v^*, a_p^* \rangle \leftarrow \langle -\infty, nil \rangle$;
  for $M_k \in$ `Subtasks`(M_i) **do**
    if M_k *is primitive* **or** $s \notin G_k$ **then**
      $\langle v, a_p \rangle \leftarrow$ `EvaluateState`(k, s, d);
      $v \leftarrow v +$ `EvaluateCompletion`(i, s, k, d);
      if $v > v^*$ **then**
        $\langle v^*, a_p^* \rangle \leftarrow \langle v, a_p \rangle$;
  return $\langle v^*, a_p^* \rangle$;

task M_0 based on the recursive Q function. Meanwhile, the true primitive action $a_p \in A$ that should be performed first can also be determined. This action a_p will be executed to the environment, leading to a transition of the system state. Then, the planning procedure starts over to select the best action for the next step.

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

3.2 Task Evaluation over Hierarchy

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

When the task hierarchy is given, it is more difficult to perform such search procedure since each subtask may contain other subtasks or several primitive actions. As shown in Algorithm 2, the search starts with the root task M_i and the current state s. Then, the node of the current state s is expanded by trying each possible subtask of M_i. This involves a recursive evaluation of the subtasks and the subtask with the highest value is selected. As mentioned in Section 2.2, the evaluation of a subtask requires the computation of the value function for its children and the completion function. The value function can be computed recursively. Therefore, the key challenge is to calculate the completion function.

Intuitively, the completion function represents the optimal value of fulfilling the task M_i after executing a subtask M_a first. According to Equation 6, the completion function of an optimal policy π^* can be written as:

$$C^{\pi^*}(i,s,a) = \sum_{s',N} \gamma^N P(s',N|s,a) V^{\pi^*}(i,s') \tag{10}$$

where

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

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

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

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

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

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

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

3.3 Completion Function Approximation

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

Notice that the term γ^N in Equation 6 is equal to 1 when $\gamma = 1$, which is the default setting of this paper. Given an optimal policy, the subtask will terminate at a certain goal state with the probability of 1 after several steps. To compute the completion function, the only term need to be considered is $P(s', N|s, a)$–a distribution over the terminal states. Given a subtask, it is often possible to directly approximate the distribution disregarding the detail of execution.

Based on these observations, we assume that each subtask M_i will terminate at its terminal states in G_i with a prior distribution of D_i. In principle, D_i can be any probability distribution associated with each subtask. Denoted by $\tilde{G}_a$ a set of sampled states drawn from prior distribution D_a using *importance sampling* [19] techniques, the completion function $C^*(i, s, a)$ can be approximated as:

$$C^*(i, s, a) \approx \frac{1}{|\tilde{G}_a|} \sum_{s' \in \tilde{G}_a} V^*(i, s'). \tag{11}$$

A recursive procedure is proposed to estimate the completion function, as shown in Algorithm 3. In practice, the prior distribution D_a–a key distribution when

Algorithm 4: `NextAction`(i, s)

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

```
if SearchStopped(i, s) then
    return nil;
else
    a* ← argmax_{a∈A_i} H_i[s, a] + c√(ln N_i[s] / N_i[s, a]);
    N_i[s] ← N_i[s] + 1;
    N_i[s, a*] ← N_i[s, a*] + 1;
    return a*;
```

computing the completion function, can be improved by considering the domain knowledge. Take the robot soccer domain for example. The agent at state s may locate in a certain position of the field. Suppose s' is the goal state of successfully scoring the ball. Then, the agent may have higher probability to reach s' if it directly dribbles the ball to the goal or passes the ball to some teammates who is near the goal, which is specified by the action a in the model.

3.4 Heuristic Search in Action Space

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

Algorithm 4 gives an UCB1 [1] version of `NextAction`, where $N_i[s]$ and $N_i[s, a]$ denote the visitation count for state s and state-action pair (s, a) respectively for subtask M_i. In Algorithm 4, $c\sqrt{\frac{\ln N_i[s]}{N_i[s,a]}}$ is a biased bonus with higher value for rarely-tried actions in order to encourage exploration on them. c is a constant variable that balances the tradeoff between exploitation and exploration. The procedure `SearchStopped` dynamically determines whether the search process for current task should be terminated based on some prune strategies. $H_i[s, a]$ is the heuristic value for action a in state s for each subtask M_i, which are initialized according to some prior domain knowledge for each subtask. They will be updated incrementally while the agent interacting with the environment according to a gradient rule, $H_i[s, a] \leftarrow H_i[s, a] + \alpha(Q(i, s, a) - H_i[s, a])$, which is commonly used in reinforcement learning algorithms [18].

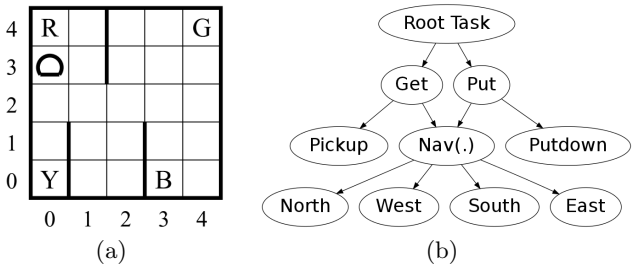

Fig. 2. (a) Taxi domain, and (b) MAXQ task graph for Taxi

4 Experiments: Taxi Domain

The Taxi domain is a common benchmark for planning and learning under uncertainty [8]. As illustrated in Figure 2(a), it consists of a 5×5 grid world with walls and 4 taxi terminals, named R, G, Y and B. The goal of the taxi agent is to pick up a and deliver a passenger. The system has 4 state variables: 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 be dl. Therefore, this problem has totally 400 states with 25 taxi locations, 5 passenger locations, and 4 destination locations, excluding those states where $pl = dl$. This is identical to the setting of [10]. At the beginning of each episode, the taxi's location, the passenger's location and the passenger's destination are all randomly chosen. It terminates when successfully delivering the passenger. There are 6 primitive actions: a) 4 navigation actions that move the agent one grid: North, South, East and West; b) the Pickup action; and c) the Putdown action. Each navigation action has the probability of 0.8 to move the agent in the indicated direction, and 0.1 for each perpendicular direction. Each action has a cost of -1, and the agent received a reward of +20 when the episode terminates with the Putdown action and a penalty of -10 for illegal Pickup and Putdown actions.

When applying MAXQ-OP to this domain, we used the same MAXQ hierarchical structure as presented in [8] shown in Figure 2(b). Note that the Nav(t) subtask takes a parameter of t with the value of either R, G, Y or B. The definition of the non-primitive subtasks is shown in Table 1. The "Irrelevant Variables" gives the state variables that can be ignored when applying state abstractions for each subtask, whilst the "Max Depth" specifies the pre-defined maximal forward search depths for each subtask used in the experiments.

The `EvaluateCompletion` procedure was implemented as follows. For high level subtasks such as Root, Get, Put and Nav(t), we assumed that they will terminate in the terminal states with the probability of 1 and for primitive subtasks such as North, South, East and West, the domain's underlying transition model $P(s, a, s')$ is used to sample the most likely state according to its transi-

Table 1. Non-primitive subtasks for 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

tion probability. For each non-primitive subtask, the `HeuristicValue` function was 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: $|x_1 - x_2| + |y_1 - y_2|$.

To speed up the search process, a cache-based strategy is implemented to prune unnecessary actions while searching in the action space of each subtask. More precisely, if state s has been evaluated for subtask M_i with depth $d[i] = 0$, assuming the result is $\langle v, a_p \rangle$, then this result will be stored in a cache table as:

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

where *cache* is the cache table, and $abstract(i, s)$ gives the abstracted form of state s for subtask M_i. Next time when the evaluation value of state s under the same condition is requested, then the cached result will be returned immediately with the probability of 0.9.

In our experiments, we ran several trials for each algorithm with a random initial state, and reported the averaged value (accumulated reward). We compared our MAXQ-OP algorithm with the R-MAXQ and MAXQ-Q algorithms presented in [10], which are the best existing MAXQ hierarchical decomposition algorithms we can find in the literature. It is not totally fair comparison since both R-MAXQ and MAXQ-Q are reinforcement learning algorithms. However, these comparisons empirically confirmed the soundness of MAXQ-OP for its ability to exploit the hierarchical structure. Additionally, we developed an asynchronous value iteration algorithm to solve the flat-represented problem optimally. This output the upper bound–the maximal value that planning algorithms can achieve–of this domain to show the quality of our approximation.

It can be seen from Table 2 that our MAXQ-OP algorithm is able to find the near-optimal policy of the Taxi domain online with the value of 3.93 ± 0.16, very close to the optimal value 4.01 ± 0.15. The value achieved by MAXQ-OP is also very competitive to R-MAXQ and MAXQ-Q. In summary, this set of experiments demonstrates the superior performance of MAXQ-OP in the Taxi benchmark domain.

Table 2. Empirical results of Taxi domain

Algorithm	Trials	Average Rewards*	Offline Time	Avg. Online Time
MAXQ-OP	1000	3.93 ± 0.16	-	0.20 ± 0.16 ms
R-MAXQ	100	3.25 ± 0.50	1200 ± 50 episodes	-
MAXQ-Q	100	0.0 ± 0.50	1600 episodes	-

*The upper bound of Average Rewards is 4.01 ± 0.15 averaged over 1000 trials.

5 Case Study: RoboCup 2D

It is our long-term effort to apply the MAXQ-OP framework to the RoboCup soccer simulation 2D domain [12, 15], a very large testbed for the research of decision-theoretic planning. In this section, we present a case-study of this domain and evaluate the performance of MAXQ-OP based on the general competition results with several high-quality teams in the RoboCup simulation 2D community. The goal is to test the scalability of MAXQ-OP and shows that it can solve large real-world problems that are previously intractable.

5.1 Introduction to RoboCup 2D

In RoboCup 2D, a central *server* simulates a 2-dimensional virtual soccer field in real-time. Two teams of fully autonomous agents connect to the server via network sockets to play a soccer game over 6000 steps. Each team consists of 11 soccer player agents, each of which interacts independently with the server by 1) receiving a set of observations; 2) making a decision; and 3) sending actions back to the server. Observations for each player only contain noisy and local geometric information such as the distance and angle to other players, ball, and field markings within its view range. Actions are atomic commands such as turning the body or neck to an angle, dashing in a given direction with certain power, kicking the ball to an angle with power, or slide tackling the ball. The key challenge lies in the fact that it is a fully distributed, multi-agent stochastic domain with continuous state, action and observation space [17]. More details about RoboCup 2D can be found at the official website.[1]

5.2 RoboCup 2D as an MDP

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

State Space We treat teammates and opponents as part of the environment and try to estimate the current state with sequences of observations. Then, the state of 2D domain can be represented as a fixed-length vector, containing state

[1] `http://wiki.robocup.org/wiki/Soccer_Simulation_League`

variables that totally cover 23 distinct objects (10 teammates, 11 opponents, the ball, and the agent itself).

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

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

Reward Function The underlying reward function has a *sparse* property: the agent usually earns zero rewards for thousands of steps before any scoring happens. If this underlying reward function is used directly, then the forward search process may often terminate without any real rewards obtained, and thus can not tell the differences between subtasks. In our team, to emphasize each subtask's characteristic and to guarantee that positive results can be found by the search process, a set of pseudo-reward functions is developed for each subtask.

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

To model the RoboCup 2D domain as an MDP which assumes that the environment's state is fully observable, the agent must overcome the difficulty that it can only receive local and noisy observations, to obtain a precise enough estimation of the environment's current state. In our team, the agent estimates the current state from its *belief* [11]. A belief b is a probability distribution over state space, with $b(s)$ denoting the probability that the environment is actually in state s. We assume conditional independence between individual objects, then the belief $b(\boldsymbol{s})$ can be expressed as

$$b(\boldsymbol{s}) = \prod_{0 \le i \le 22} b_i(\boldsymbol{s}[i]), \tag{12}$$

where $\boldsymbol{s}$ is the full state vector, $\boldsymbol{s}[i]$ is the partial state vector for object i, and $b_i(\boldsymbol{s}[i])$ is the marginal distribution for $\boldsymbol{s}[i]$. A set of m_i weighted samples (also known as particles) are then used to approximate b_i as:

$$b_i(\boldsymbol{s}[i]) \approx \{\boldsymbol{x}_{ij}, w_{ij}\}_{j=1 \ldots m_i}, \tag{13}$$

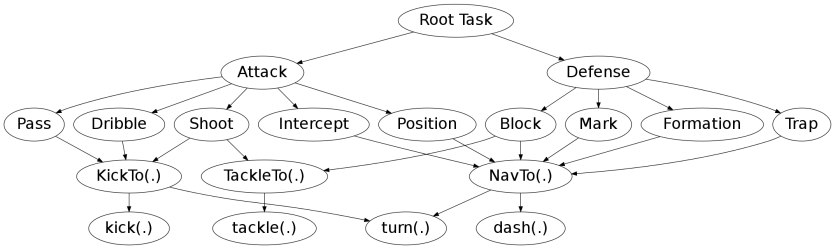

Fig. 3. MAXQ task graph for our team

where $\boldsymbol{x}_{ij}$ is a sampled state for object i, and w_{ij} represents the approximated probability that object i is in state $\boldsymbol{x}_{ij}$ (obviously $\sum_{1 \le j \le m_i} w_{ij} = 1$).

In the beginning of each step, these samples are updated by Monte Carlo procedures using the domain's *motion model* and *sensor model* [7]. Finally, the environment's current state $\boldsymbol{s}$ is estimated as:

$$\boldsymbol{s}[i] = \sum_{1 \le j \le m_i} w_{ij} \boldsymbol{x}_{ij}. \tag{14}$$

Based on empirical results taken from actual competitions, the estimated state is sufficient for the agent to make good decisions, particularly for the state of the agent itself and other close objects.

5.3 Solution with MAXQ-OP

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

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

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

The graphical representation of the MAXQ hierarchical structure is shown in Figure 3, where a parenthesis after a subtask's name indicates this subtask will take parameters. It is worth noting that state abstractions are implicitly introduced by this hierarchy. For example in the NavTo subtask, only the agent's own state variables are relevant. It is irrelevant for the KickTo and TackleTo subtasks to consider those state variables describing other players' states. To deal with the large action space, heuristic methods are critical when applying MAXQ-OP. There are many possible candidates depending on the characteristic of subtasks. For instance, hill-climbing is used when searching over the action space of KickTo for the Pass subtask and A* search is used when searching over the action space of dash and turn for the NavTo subtask.

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

$$impelling\_speed(s, s', \alpha) = \frac{dist(s, s', \alpha) + pre\_dist(s', \alpha)}{step(s, s') + pre\_step(s')} \tag{15}$$

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

5.4 Empirical Evaluation

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

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

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

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

The selected scene, which is originally located at cycle #3142 of that match, is depicted in Figure 4 where white circles represent our players, gray ones represent opponents, and the small black one represents the ball. We can see that our player 10 was holding the ball at that moment, while 9 opponents (including goalie) were blocking just in front of their goal area. In RoboCup 2011, our player 10 passed the ball directly to teammate 11. Having got the ball, our player 11 decided to pass the ball back to teammate 10. When teammate 11 had moved to an appropriate position, the ball was passed again to it. Finally, teammate 11 executed a tackle to shoot at cycle #3158 and scored a goal 5 cycles later.

Table 3 summarizes the test results showing that the FULL version of our team outperforms both RANDOM and HAND-CODED with an increase of the chance of **sucess** by 87% and 65% respectively. We find that although FULL, RANDOM and HAND-CODED have the same hierarchical structure and subtasks of Attack, the local selection strategy between Pass and Dribble plays a key role in the decision of Attack and affects the final performance substantially. It can be seen from the table that MAXQ-OP based local selection strategy between Pass and Dribble is sufficient for the Attack behavior to achieve a high performance. Recursively, this is also true for other subtasks over the MAXQ hierarchy, such

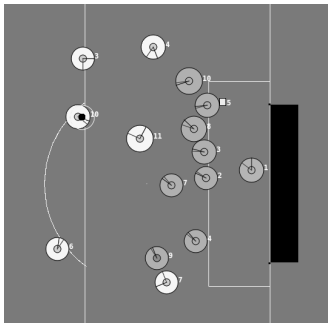

Fig. 4. A selected scene from a real match in RoboCup 2011

Table 3. Empirical results of our team in episodic scene test

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

as Defense, Shoot, Pass, etc. To conclude, MAXQ-OP is able to be the key to sucess of our team in this episodic scene test.

We also tested the FULL version of our team in full games against 4 high-quality RoboCup 2D opponent teams, namely BrainsStomers08, Helios10, Helios11 and Oxsy11. Notice that BrainStormers08 and Helios10 were the champion of RoboCup 2008 and RoboCup 2010 respectively. In the experiments, we independently ran our team against the binary codes officially released by them for 300 games on exactly the same hardware. Table 4 summarizes the detailed empirical results with our winning rate, which is defined as $p = n/N$, where n is the number of games we won, and N is the total number of games. It can be seen from the table that our team with the implementation of MAXQ-OP substantially outperforms other tested teams. Specifically, our team had about 82.16%, 93.15%, 83.33% and 91.32% of the chances to win BrainsStomers08, Helios10, Helios11 and Oxsy11 respectively. Table 5 summarizes the general performance of our team with MAXQ-OP in the RoboCup competitions of past 7 years. Competition logfiles can be found at the official website.[2]

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

[2] `http://ssl.robocup-federation.org/ftp/2d/log/`

Table 4. Empirical results of our team in full games

Opponent Team	Games	Avg. Goals	Avg. Points	Winning Rate
BrainsStomers08	300	3.09 : 0.82	2.59 : 0.28	$82.16 \pm 4.33\%$
Helios10	300	4.30 : 0.88	2.84 : 0.11	$93.15 \pm 2.86\%$
Helios11	300	3.60 : 1.09	2.60 : 0.30	$83.33 \pm 4.22\%$
Oxsy11	300	4.97 : 1.33	2.79 : 0.16	$91.32 \pm 3.19\%$

Table 5. History results of our team in RoboCup annual competitions

Competitions	Games	Points	Goals	Win	Draw	Lost	Avg. Points	Avg. 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

1) it demonstrates the scalability and efficiency of MAXQ-OP for solving a large real-world application such as RoboCup soccer simulation 2D; 2) it presents a decision-theoretic solution for developing a RoboCup soccer team, which is more general and easy for programming high-level strategies.

6 Conclusions

This paper presents MAXQ-OP–a novel online planning algorithm that benefits from both the advantage of hierarchical decomposition and the power of heuristics. It recursively expands the AND-OR tree online and searches over the policy space by following the pre-defined task hierarchy. This is more efficient since only relevant states and actions are considered according to the MAXQ hierarchy. Another contribution of this work is approximate the prior distribution when computing the completion function. The key observation is that the prior distribution can be specified based on the task hierarchy as well as the domain knowledge of this task. By given such prior distributions, MAXQ-OP can evaluate the root task online without actually computing the sub-policy for every subtask. Similar to our work, Barry et al. proposed an *offline* algorithm called DetH* [3] to solve large MDPs hierarchically by assuming that the transitions between macro-states are totally deterministic. In contrast, we assume a prior distribution over the terminal states of each subtask, which is more realistic. The empirical results show that MAXQ-OP is able to find a near-optimal policy online for the Taxi domain and solve a very large problem such as the RoboCup 2D that are previously intractable in the literature of the decision-theoretic planning. This demonstrates the soundness and stability of MAXQ-OP for solving large MDPs with the pre-defined task hierarchy. In the future, we plan to theo-

retically analyze MAXQ-OP with different task priors and try to generate these priors automatically.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.
2. J. Barry. *Fast Approximate Hierarchical Solution of MDPs*. PhD thesis, Massachusetts Institute of Technology, 2009.
3. J. Barry, L. Kaelbling, and T. Lozano-Perez. Deth*: Approximate hierarchical solution of large markov decision processes. In *International Joint Conference on Artificial Intelligence*, pages 1928–1935, 2011.
4. A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.
5. D. Bertsekas. *Dynamic programming and optimal control*. Athena Scientific, 1996.
6. B. Bonet and H. Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *International Conference on Automated Planning and Scheduling*, volume 3, 2003.
7. F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1322–1328. IEEE, 2001.
8. T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Machine Learning Research*, 13(1):63, May 1999.
9. E. Hansen and S. Zilberstein. Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.
10. N. Jong and P. Stone. Hierarchical model-based reinforcement learning: R-max + MAXQ. In *Proceedings of the 25th international conference on Machine learning*, pages 432–439. ACM, 2008.
11. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.
12. H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada. The robocup synthetic agent challenge 97. In *International Joint Conference on Artificial Intelligence*, volume 15, pages 24–29, 1997.
13. L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. *European Conference on Machine Learning*, pages 282–293, 2006.
14. M. Littman, T. Dean, and L. Kaelbling. On the complexity of solving markov decision problems. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 394–402. Citeseer, 1995.
15. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2-3):233–250, 1998.
16. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.
17. P. Stone. *Layered learning in multiagent systems: A winning approach to robotic soccer*. The MIT press, 2000.
18. R. Sutton and A. Barto. *Reinforcement learning: An introduction*, volume 116. Cambridge Univ Press, 1998.
19. S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust monte carlo localization for mobile robots. *Artificial intelligence*, 128(1-2):99–141, 2001.