Neural Comput & Applic (2015) 26:775–787
DOI 10.1007/s00521-014-1738-2

ORIGINAL ARTICLE

# A data-based online reinforcement learning algorithm satisfying probably approximately correct principle

**Yuanheng Zhu · Dongbin Zhao**

Received: 14 May 2014 / Accepted: 14 September 2014 / Published online: 30 October 2014

**Abstract** This paper proposes a probably approximately correct (PAC) algorithm that directly utilizes online data efficiently to solve the optimal control problem of continuous deterministic systems without system parameters for the first time. The dependence on some specific approximation structures is crucial to limit the wide application of online reinforcement learning (RL) algorithms. We utilize the online data directly with the kd-tree technique to remove this limitation. Moreover, we design the algorithm in the PAC principle. Complete theoretical proofs are presented, and three examples are simulated to verify its good performance. It draws the conclusion that the proposed RL algorithm specifies the maximum running time to reach a near-optimal control policy with only online data.

**Keywords** Reinforcement learning · Probably approximately correct · Kd-tree

## 1 Introduction

The online reinforcement learning (RL) draws a lot of attention both from the computer science [1–4] and from the optimal control science [5–7], because it uses the online data to achieve an optimal policy through the interaction with the environment. Compared to the offline RL, the efficient usage of online data or the trade-off of exploration and exploitation becomes more critical. Besides, some issues related to practical implementation, e.g., the convergence rate and the obtained optimality, also need our consideration. Lots of efforts [8–16] have been devoted to solve such problems from different aspects.

Among many studies to overcome these problems, the probably approximately correct (PAC) is one of the most effective approaches. In the running process of an online learning algorithm, if the sum of steps when it implements non-optimal actions is finite and bounded, then it is called a PAC algorithm. Considering finite Markov Decision Processes (MDPs) with finite states, a lot of PAC algorithms have been proposed, including $E^3$ [17], RMAX [18], MBIE [19], Delayed Q-learning [20].

For recent years, many researchers have concentrated on continuous-state systems to solve online optimal control problems in the PAC principle. These include Kakade et al. with their Metric-$E^3$ [21] and Pazis and Parr with their C-PACE [22]. However, time bounds (number of steps that implement non-optimal actions) for these algorithms are proved to be polynomial and finite only with probabilities. This means with some possibilities, it fails to draw their conclusions. Meanwhile, Bernstein and Shimkin [23] propose the ARL algorithm for continuous deterministic systems. They prove their algorithm has a determinate finite time bound. But, the implementation requires some parameters of systems. So, it is partially dependent on system information, which limits its application.

In this paper, we consider the optimal control problem of continuous deterministic systems and propose an online data-based RL algorithm. Without relying on any specific approximation structure, the online data are used directly. A kd-tree technique is adopted. The implementation is based on the current collected data and is applicable for arbitrary control problems.

Y. Zhu · D. Zhao (✉)
The State Key Laboratory of Management and Control for Complex Systems, Institution of Automation, Chinese Academy of Sciences, Beijing, China
e-mail: dongbin.zhao@ia.ac.cn

Y. Zhu
e-mail: yuanheng.zhu@gmail.com

As we adopt kd-tree and consider the PAC principle on continuous-state systems, we term our algorithm as kd-CPAC. The major contributions are summarized as follows. The implementation of kd-CPAC is not dependent on any specific approximation structure and is totally based on data. Through analysis, it is proved that kd-CPAC satisfies the PAC principle. This conclusion helps to infer that for certain online cases, the maximum running time to reach a near-optimal policy is bounded. It is the first implementation of a PAC algorithm that directly utilizes samples to solve continuous deterministic systems without any system parameters.

The paper is organized as follows. Section 2 introduces the background for RL, and Sect. 3 describes the kd-tree technique adopted in the algorithm. The whole process of kd-CPAC algorithm and the theoretical results are presented in Sect. 4. Proofs of lemmas and theorems are given in Sect. 5. We simulate three examples in Sect. 6 to verify the performance of our algorithm. The end is our discussion and conclusion.

## 2 Formulation of online reinforcement learning

A continuous-state system with deterministic transition function can be represented by a four-tuple, ($S$, $A$, $R$, $F$). $S$ is an $n$-dimensional continuous-state space, $A$ denotes a discrete action set, $R(s, a)$ is the reward function, and $F(s, a)$ is the deterministic transition function that indicates the next-step state at state-action pair ($s$, $a$). Suppose the state space is bounded, not infinitely extended. The reward function also has an interval, namely $r_{\min} \le R(s,a) \le r_{\max}$. Note that in this case, $R$ and $F$ are both unknown to algorithms. So, the only available information is online observations ($s$, $a$, $r$, $s'$), where $r$ is the received reward and $s'$is the next-step state at ($s$, $a$).

During the interaction with the environment, we assume that at time $t$, the agent has experienced a history of states and actions, denoted by

$$h_t = \{s_0, a_0, s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t\}.$$

In the online case, the policy is *non-stationary* as algorithms can modify it at any moment. So, actions are selected following a series of policies $\pi = \{\pi_t\}_{t=0}^{\infty}$, namely $a_t = \pi_t(s_t)$.

To evaluate the performance of a policy, we adopt the *discounted return criterion*. Given a policy $\pi$ and an initial state $s_0 = s$, the discounted return is defined as

$$J^{\pi}(s) \triangleq \sum_{t=0}^{\infty} \gamma^t r_t \big|_{s_0=s, a_t=\pi_t(s_t)} \tag{1}$$

where $\gamma$ is the *discounted factor* satisfying $0 < \gamma < 1$. Note that in some systems, agents may stop and get stuck at some *terminal states*. Then, the discounted return in this case is modified to

$$J^{\pi}(s) \triangleq \sum_{t=0}^{T-1} \gamma^t r_t + V(s_{\mathrm{T}}) \big|_{s_0=s, a_t=\pi_t(s_t)} \tag{2}$$

where $s_{\mathrm{T}}$ indicates the terminal state and $Vs_{\mathrm{T}}$ is a predefined value to estimate the success or failure at $s_{\mathrm{T}}$. We see (2) as a special case of (1) and only consider the first expression below.

The target of RL is maximizing the value of $J^{\pi}(s)$, namely finding the *optimal value function*, defined by $V(s) \triangleq \max_{\pi} J^{\pi}(s)$. The corresponding policy is called the *optimal policy*, $\pi^* \triangleq \arg\max_{\pi} J^{\pi}$. Sometimes, the optimal policy is too difficult to obtain, and then, a *near-optimal* policy with the similar performance is acceptable. Policy $\pi$ is called $\varepsilon$-optimal if $J^{\pi}(s) \ge V(s) - \varepsilon$ holds for all $s \in S$.

In some cases, *optimal action-value function* or *optimal Q function* is preferred for the implementation of RL. It is defined as $Q(s,a) \triangleq r + \gamma V(s')$ and can be expressed in a Bellman principle

$$Q(s,a) = r + \gamma \max_{a'} Q(s', a'),$$

and the optimal policy is generated by

$$\pi^*(s) = \arg\max_{a} Q(s,a).$$

Similarly, $V$ is obtained from $Q$ by $V(s) = \max_{a} Q(s,a)$. In the following sections, our interest is mainly focused on the form of $Q$ function.

As the reward function is bounded in an interval $[r_{\min}, r_{\max}]$, so the discounted return has a lower bound and an upper bound, denoted by $\frac{r_{\min}}{1-\gamma}$ and $\frac{r_{\max}}{1-\gamma}$, respectively. Furthermore, define $V_{\mathrm{b}} \triangleq \frac{1}{1-\gamma}(r_{\max} - r_{\min})$ as the maximum difference of returns between any two policies.

To evaluate the performance of an online algorithm, a definition of *policy-mistake count* (PMC) is introduced, which specifies the sum of steps when the algorithm implements non-optimal actions during its whole process of online running.

**Definition 1** *(PMC)* [23] In an algorithm, $h_t$ is its history of states and actions and $A_t = \{\pi_k\}_{k=t}^{\infty}$ is the policy that the algorithm implements at time $t$. The discounted return of $A_t$ from $t$ is denoted by $J^{A_t}(s_t) \triangleq \sum_{k=t}^{\infty} \gamma^{(k-t)} r_k \big|_{a_k=\pi_k(s_k)}$. So, the policy-mistake count is defined as

$$\mathrm{PMC}(\varepsilon) \triangleq \sum_{t=0}^{\infty} \mathbb{I}\{J^{A_t}(s_t) < V(s_t) - \varepsilon\}$$

where $\mathbb{I}\{\cdot\}$ is a signal function. If the event in brace occurs, it outputs 1, otherwise 0.

So, at the other time excluding the PMC steps, the algorithm has near-optimal performance at corresponding states. If an algorithm's PMC is finite and bounded, the algorithm is called PAC.

As the information of systems is only from online observations, the storage of these samples is a major problem in the algorithm. Next, we will give a brief description about kd-tree, which is used to store online samples.

## 3 Kd-tree for the storage of samples

Kd-tree, as an efficient approach to split the state space and store data, has been applied widely in the field of RL. For example, Munos and Moore [24] use kd-trie, a special version of kd-tree, to accomplish a variable resolution discretization of state space for dynamic programming. Ernst et al [25] study the kd-tree-based regression algorithm. Here, kd-tree is adopted to store the online samples. Details about kd-tree technique are available in [26]

Suppose a sample is denoted by $(\hat{s}, \hat{a}, \hat{r}, \hat{s}')$. We take $\hat{s}$ as the key to refer the sample. For convenience, each action corresponds to a kd-tree and there are $|A|$ kd-trees–$|A|$ indicates the number of actions. At the beginning, each tree has an empty root, which occupies the whole state space. When samples arrive, they are stored in the root. When the number of stored samples reaches a maximum number $N_{\text{split}}$—*split condition*, the space of the root is split into two nodes by a *split hyperplane* at the *split dimension*. Therefore, the $N_{\text{split}}$ samples are divided equally into two children. This process will continue if more samples arrive and the depth of the tree will increase larger and larger.

The split dimension and hyperplane are determined by the following principle. Calculate the variance of each dimension among the samples in the node which is going to be split. For better comparison, states are normalized by the span of the state space before the calculation. The dimension corresponding to the maximum variance is selected as the split dimension. Then, choose the median value at this dimension among samples as the split hyperplane.

When a new sample $(\hat{s}, \hat{a}, \hat{r}, \hat{s}')$ is required to be added in kd-trees, search the kd-tree of action $\hat{a}$ for the leaf which $\hat{s}$ belongs to. Then, put the sample in the leaf node.

For arbitrary states, it is also convenient to find their neighboring samples in kd-trees. Given a state $s$, the *neighboring samples* of $s$ refer to the set of samples $(\hat{s}, \hat{a}, \hat{r}, \hat{s}')$ which satisfy $d(s, \hat{s}) \leq \delta$. $d$ is a metric $d : S \times S \to \mathbb{R}$ to specify the distance between two states, and $\delta$ is the *neighboring distance*. Start from the root and estimate the area of each node if it is close to $s$ within $\delta$. If not, its children and the included samples are also farther away than $\delta$ and there is no need to consider them. Otherwise, if the distance between the node and $s$ is less than $\delta$, and the node has children, then continue to estimate each child in the same way until reaching leaves. Compare the samples in the leaves with $s$ and output those whose distances are less than $\delta$ as neighboring samples.

Based on the principle of storing samples with kd-tree, we present the kd-CPAC algorithm in the following section, whose theoretical proof is given in Sect. 5.

## 4 Kd-tree-based continuous PAC algorithm

Before introducing the kd-CPAC algorithm, some assumptions about the continuity of systems are required.

**Assumption 1** *(Continuity)* [23] For any $s_1, s_2 \in S$ and $a \in A$, there exist two constants $\alpha$ and $\beta$ such that

$$|R(s_1, a) - R(s_2, a)| \leq \alpha d(s_1, s_2)$$
$$d(F(s_1, a), F(s_2, a)) \leq \beta d(s_1, s_2).$$

$\alpha$ and $\beta$ are called *continuity constants* of reward and transition functions, respectively.

Based on Assumption 1, a lemma about the optimal Q function is deduced.

**Lemma 1** *For any $s_1, s_2 \in S$ and $a \in A$, we have*

$$|Q(s_1, a) - Q(s_2, a)| \leq \bar{\omega}(d(s_1, s_2))$$

*where $\bar{\omega}$ is defined as*

1. *If $\gamma\beta < 1$*
$$\bar{\omega}(z) = \frac{\alpha}{1 - \gamma\beta} z$$
2. *If $\gamma\beta > 1$*
$$\bar{\omega}(z) = c z^{\log_\beta(1/\gamma)}$$
where
$$c \triangleq 2\beta \left( \frac{\alpha}{\gamma\beta - 1} \right)^{\log_\beta(1/\gamma)} V_{\text{b}}^{\log_\beta(\gamma\beta)}.$$

The above lemma is about the continuity of $Q(s, a)$. For $\bar{\omega}$, if $z \to 0$, $\bar{\omega}(z) \to 0$. Note that a similar result is available in [23], while the original statement refers to $V(s)$. But the mechanism of the proofs is almost the same, so we omit it here.

Note that $\alpha$, $\beta$, and $\bar{\omega}$ are all parameters and function about systems. In general, they are unknown to algorithms. Next, we will present a detailed description of our algorithm. In its implementation, no knowledge of these values is required. But in the theoretical analysis, these parameters and function will help to obtain our lemmas and theorems.

### 4.1 Data set

Based on the previous section, suppose the current time is $t$ and we have a data set $D_t = \{(\hat{s}_i, \hat{a}_i, \hat{r}_i, \hat{s}'_i)\}$ stored in kd-trees, in which are selective samples of the past time. As $\hat{r}_i$ and $\hat{s}'_i$ are determined by $\hat{s}_i$ and $\hat{a}_i$, so we can simplify the expression of a sample by the pair $(\hat{s}_i, \hat{a}_i)$. Or just $\hat{s}_i$ if given $\hat{a}_i$.

Use $D_t(a)$ to represent the set of samples belonging to $a$. Furthermore, for an arbitrary state $s$ at action $a$, we construct a *neighboring set*—$C_t(s,a)$, which includes the neighboring samples $(\hat{s}_i, a, \hat{r}_i, \hat{s}'_i)$ in $D_t(a)$ that have $d(s, \hat{s}_i) \le \delta$, where $\delta$ is the predefined neighboring distance. If there exists no such samples, we say $C_t(s,a) = \emptyset$. So, $C_t(s,a)$ indicates the set of samples that is $\delta$-close to $s$ and we can further use them to approach $(s, a)$. Moreover, for any pair $(s, a)$ and its arbitrary neighboring sample $(\hat{s}_i, a, \hat{r}_i, \hat{s}'_i)$, they have the following inequalities based on Assumption 1

$$|R(s,a) - R(\hat{s}_i, a)| \le \alpha\delta \tag{3}$$

$$d(F(s,a) - F(\hat{s}_i, a)) \le \beta\delta. \tag{4}$$

### 4.2 Data-based Q-Iteration

Based on the data set, we can utilize it to define a *Data-Based Q-Iteration* (DBQI) operator.

**Definition 2** *(DBQI operator)* Given a function $g : S \times A \to \mathbb{R}$ and arbitrary $s$ and $a$, the DBQI operator $\mathcal{T}$ is defined as

$$\mathcal{T}(g)(s,a) = \begin{cases} \min\limits_{\hat{s}_i \in C_t(s,a)} \left[\hat{r}_i + \gamma \max\limits_{a'} g(\hat{s}'_i, a')\right], & \text{if } C_t(s,a) \ne \emptyset \\ V_{\max}, & \text{otherwise} \end{cases} \tag{5}$$

where $(\hat{s}_i, a, \hat{r}_i, \hat{s}'_i)$ denotes the neighboring samples of $(s, a)$ if $C_t(s,a)$ is not empty.

DBQI operator means for a pair $(s, a)$, if its $C_t(s,a)$ is empty, we assign the value of $(s, a)$ with the upper bound of value function, $V_{\max}$. Otherwise, we use the neighboring samples in $C_t(s,a)$ to approach its value, more concretely, the minimum one corresponding to the right side of the equation in (5). It is obvious that the calculation of DBQI operator is totally based on the stored samples.

We can prove $\mathcal{T}$ is a contraction operator, so there exists a fixed solution that has $\hat{Q}_t = \mathcal{T}(\hat{Q}_t)$. $\hat{Q}_t$ is called *Data-Based Q Function* (DBQF). To calculate $\hat{Q}_t$, value iteration (VI) [27, 28] or policy iteration (PI) [29, 30] can be used. In Appendix, we present a simple and convenient VI method to calculate it. The only need in the method is to calculate $Q$ values of stored samples, and then, the exact $\hat{Q}_t$ over the whole state space is obtained.

About $\mathcal{T}$ and $\hat{Q}_t$, we have two lemmas as follows and the corresponding proofs are presented in the next section.

**Lemma 2** *The operator $\mathcal{T}$ is a contraction with the factor $\gamma$ in the infinity norm.*

**Lemma 3** *For every time $t$, $\hat{Q}_t$ has the following relationship with respect to the optimal $Q$ function*

$$\hat{Q}_t(s,a) \ge Q(s,a) - \frac{\alpha\delta + \gamma\varpi(\beta\delta)}{1-\gamma}, \quad \forall s \in S \text{ and } \forall a \in A.$$

With $\hat{Q}_t$, a greedy policy is extracted and applied to the system online to obtain a new observation at the next step

$$\pi_t(s) = \arg\max_a \hat{Q}_t(s,a). \tag{6}$$

### 4.3 Escape event

The next issue is whether to add the new observation into $D_t$ or not. At the beginning of the algorithm, $D_0$ is empty. As the implementation progresses, some observations are added in $D_t$, while some are ignored to avoid the data set increasing infinitely. The principle is only storing the samples that have useful information about systems. So, a definition of known is given here.

**Definition 3** *(Known)* Given an observation $(s, a, r, s')$. If $C_t(s,a) \ne \emptyset$ and there exists a sample $(\hat{s}_i, a_i, \hat{r}_i, \hat{s}'_i) \in C_t(s,a)$ such that $\left|\max\limits_{a'_1} \hat{Q}_t(s', a'_1) - \max\limits_{a'_2} \hat{Q}_t(\hat{s}'_i, a'_2)\right| \le \varepsilon_K$, then the observation is *known*. Otherwise, we say it is *unknown*. The parameter $\varepsilon_K$ is called *known error*.

Based on Definition 3, we have the following lemma about $C_t(s,a)$.

**Lemma 4** *For any $s$ and $a$, $C_t(s,a)$ can include at most $\lfloor V_b/\varepsilon_K \rfloor$ samples. We denote this value by $N_c$.*

When a new observation arrives, we determine whether it is known or unknown first. If known, we regard it with no useful information for our algorithm. If unknown, the observation contains some knowledge we have not encountered. Then, it is added into the data set and $D_t \to D_{t+1}$. Update $\hat{Q}_t$ and $\pi_t$ to $\hat{Q}_{t+1}$ and $\pi_{t+1}$. To denote this process, an *escape event* is defined. Before that, we introduce a common definition about the horizon time.

**Definition 4** *($\varepsilon_H$-HorizonTime)* A $\varepsilon_H$-horizon time $T_{\varepsilon_H}$ is defined as

$$T_{\varepsilon_H} \triangleq \log_{1/\gamma} \frac{V_b}{\varepsilon_H}.$$

This definition indicates that the rewards after $T_{\varepsilon_H}$ steps only have at most $\varepsilon_H$ influence to the current return.

**Definition 5** *(Escape event)* At time $t$, the system starts from a state $s$ and $A_t = \{\pi_k\}_{k=t}^{\infty}$ is the policy implemented by the algorithm. An escape event is defined as

$$\begin{aligned} E_t(s) \triangleq \{&\text{starting a trial from } s \text{ and following policy} \\ &A_t, \text{a pair } (s_\tau, a_\tau) \text{ is encountered within } T_{\varepsilon_H} \\ &\text{steps, such that } (s_\tau, a_\tau) \text{ is unknown in } D_t, \\ &\text{where } t \le \tau \le t + T_{\varepsilon_H} - 1\} \end{aligned}$$

whenever an escape event occurs, the data set is increased and the data-based $Q$ function is updated. The next lemma reflects a relationship between the return of $A_t$ and $\hat{Q}_t$ associated with the escape event.

**Lemma 5** *For every time $t$,*

$$\max_a \hat{Q}_t(s_t, a) - J^{A_t}(s_t) \le \mathbb{I}\{E_t(s_t)\} V_b + \frac{\alpha\delta}{1-\gamma} + \frac{\gamma}{1-\gamma}\varepsilon_K + \varepsilon_H$$

### 4.4 Main theorem

Before we present our main results, another definition is introduced which will be used in our theorems.

**Definition 6** *(Largest minimal cover)* [21] In a whole state space, a $\delta$-cover is a set of points with the property that for any state $s$, there exists a point $\hat{s}$ in the set satisfying $d(s, \hat{s}) \le \delta$. Let $N_\delta$ be the size of the largest minimal $\delta$-cover—that is the largest $\delta$-cover set that the removal of any point in it could render the set no longer $\delta$-cover.

**Theorem 1** *In our kd-CPAC algorithm, its PMC is bounded by*

$$\mathrm{PMC}(\varepsilon) \le N_\delta N_C |A| \log_{1/\gamma} \frac{V_b}{\varepsilon_H}$$

*where* $\varepsilon = \frac{2\alpha\delta + \gamma\bar{\omega}(\beta\delta)}{1-\gamma} + \frac{\gamma}{1-\gamma}\varepsilon_K + \varepsilon_H$.

So, during the whole process of our algorithm, the sum of steps when kd-CPAC implements non-optimal actions is finite and bounded. Therefore, kd-CPAC is a PAC algorithm. In particular, this theorem helps to deduce a result that bounds the running time for some online cases to reach a near-optimal policy.

**Theorem 2** *For an online problem, suppose it has an initial state $s_0$ and each episode has a fixed $T_{\text{episode}}$ length. At the beginning of each episode, the system is set to $s_0$, and after $T_{\text{episode}}$ steps, the episode ends and the state is reset. Apply kd-CPAC algorithm to the system. The maximum number of episodes for kd-CPAC to stop and output a near-optimal policy is bounded by $N_\delta N_C |A|$ (corresponding to $N_\delta N_C |A| T_{\text{episode}}$ steps).*

Therefore, the maximum running time of our algorithm to achieve a near-optimal policy is finite and deterministic in some parameters about systems and kd-tree. Note that the supposition about the online problem is quite common in practice. Besides, the optimal error $\varepsilon$ of the learned policy is related to neighboring distance $\delta$, the known error $\varepsilon_K$, and the horizon time error $\varepsilon_H$. If these parameters are set smaller values, the final learned policy is more optimal. But on the contrary, this will increase the running time because $N_\delta$ and $N_C$ are increased. However, in the theorems, $N_\delta N_C$ is the number of samples stored in data set at the worst case. While in practice, when the algorithm stops, the actual stored samples are far less than $N_\delta N_C$.

Another point that needs more attention is the final learned policy. From the result of Theorem 1, the policy is near-optimal only for the states along the trajectory starting from the initial state $s_0$ within $T_{\text{episode}}$ length of the episode. For the other area of state space, the optimality is not concerned. However, for online problems, the main issue is controlling systems from the start with a near-optimal performance. So here, it is simplified to declare the final policy is near-optimal.

**Algorithm 1** Kd-CPAC Algorithm

```
Require: value function upper bound $V_{\max}$
     $|A|$ kd-trees
     neighboring distance $\delta$
     known error $\varepsilon_K$
 1: initialize $D_0 \leftarrow \varnothing$, $\hat{Q}_0 \leftarrow V_{\max}$ and $\pi_0(s) = \arg\max_a \hat{Q}_0(s,a)$
 2: for $t = 0, 1, 2, \ldots$ do
 3:    observe $(s_t, a_t, r_t, s'_t)$
 4:    if $(s_t, a_t)$ is unknown in $D_t$ then
 5:       $(s_t, a_t, r_t, s'_t)$ is added into $D_t$
 6:       update $\hat{Q}_t$ according to (5)
 7:       produce $\pi_t$ according to (6)
 8:    end if
 9:    execute $\pi_t$ on the system
10: end for no change of $D_t$ happens in an episode
```

The whole process of kd-CPAC is presented in Algorithm 1. Note that no parameters about systems are involved.

## 5 Theoretical proof

First, we give the proofs for Lemma 2 and Lemma 3.

*Proof* *(Lemma 2)* Given two functions $g_1$ and $g_2$, for any $s$ and $a$, we have,

1. If $C_t(s,a) = \varnothing$, then $|\mathcal{T}(g_1)(s,a) - \mathcal{T}(g_2)(s,a)| = 0$.
2. If $C_t(s,a) \ne \varnothing$, then

$$\begin{aligned}
&|\mathcal{T}(g_1)(s,a) - \mathcal{T}(g_2)(s,a)| \\
&\quad \le \max_{\hat{s}_i \in C_t(s,a)} \left| \hat{r}_i + \gamma \max_{a'_1} g_1(\hat{s}'_i, a'_1) \right. \\
&\qquad \left. - \hat{r}_i - \gamma \max_{a'_2} g_2(\hat{s}'_i, a'_2) \right| \\
&\quad \le \gamma \max_{\hat{s}_i \in C_t(s,a)} \max_{a'} |g_1(\hat{s}'_i, a') - g_2(\hat{s}'_i, a')| \\
&\quad \le \gamma \|g_1 - g_2\|_\infty.
\end{aligned}$$

To summarize, $\|\mathcal{T}(g_1) - \mathcal{T}(g_2)\|_\infty \le \gamma \|g_1 - g_2\|_\infty$ is satisfied for any $s$ and $a$. The proof is completed. □

*Proof (Lemma 3)* With Lemma 2, we know $\hat{Q}_t$ is the fixed solution of operator $\mathcal{T}$. Here, we divide the proof of Lemma 3 into two cases. For any state $s$ and action $a$, their four-tuple is denoted by ($s$, $a$, $r$, $s'$).

1. If $C_t(s,a) = \emptyset$, then $\hat{Q}_t(s,a) = V_{\max} \ge Q(s,a)$.
2. If $C_t(s,a) \ne \emptyset$, then

$$\begin{aligned}
&\hat{Q}_t(s,a) - Q(s,a) \\
&\quad = \hat{r}_{\min} + \gamma \max_{a'_1} \hat{Q}_t(\hat{s}'_{\min}, a'_1) \\
&\qquad - r - \gamma \max_{a'_2} Q(s', a'_2) \\
&\quad \ge -\alpha\delta + \gamma \min_{a'} [\hat{Q}_t(\hat{s}'_{\min}, a') - Q(s', a')] \\
&\quad = -\alpha\delta + \gamma [\hat{Q}_t(\hat{s}'_{\min}, a'_{\min}) - Q(s', a'_{\min})].
\end{aligned}$$

In the first and last equalities, for simplicity, we denote $(\hat{s}_{\min}, a, \hat{r}_{\min}, \hat{s}'_{\min})$ and $a'_{\min}$ to specify the minimum sample and the minimum action. By Lemma 1, we know that in the optimal $Q$ function,

$$\begin{aligned}
\left| Q(\hat{s}'_{\min}, a'_{\min}) - Q(s', a'_{\min}) \right| &\le \varpi(d(\hat{s}'_{\min}, s')) \\
&\le \varpi(\beta\delta)
\end{aligned}$$

is satisfied, where the second inequality follows (4). Then, the following is further deduced

$$\begin{aligned}
&\hat{Q}_t(s,a) - Q(s,a) \\
&\quad \ge -\alpha\delta - \gamma\varpi(\beta\delta) \\
&\qquad + \gamma[\hat{Q}_t(\hat{s}'_{\min}, a'_{\min}) - Q(\hat{s}'_{\min}, a'_{\min})].
\end{aligned}$$

It is obvious that the result of $\hat{Q}_t(\hat{s}'_{\min}, a'_{\min}) - Q(\hat{s}'_{\min}, a'_{\min})$ also follows the two cases.

Thus, proceeding iteratively, we can conclude a low bound for Lemma 3:

$$\begin{aligned}
\hat{Q}_t(s,a) - Q(s,a) &\ge \sum_{t=0}^{\infty} \gamma^t (-\alpha\delta - \gamma\varpi(\beta\delta)) \\
&= -\frac{\alpha\delta + \gamma\varpi(\beta\delta)}{1-\gamma}.
\end{aligned}$$

□

Now, let us prove the property of neighboring set in Lemma 4.

*Proof (Lemma 4)* In our algorithm, an observation can be added into data set only if it is unknown. So, according to the definition, for any two samples in $C_t(s,a)$, $(\hat{s}_i, a, \hat{r}_i, \hat{s}'_i)$, and $(\hat{s}_j, a, \hat{r}_j, \hat{s}'_j)$, they must have

$$\left| \max_{a'_1} \hat{Q}_t(\hat{s}'_i, a'_1) - \max_{a'_2} \hat{Q}_t(\hat{s}'_j, a'_2) \right| > \varepsilon_{\mathrm{K}}.$$

Meanwhile, the range of $Q$ value is bounded by $V_{\mathrm{b}}$. So, the number of samples that can be stored in $C_t(s,a)$ is bounded by $\lfloor V_{\mathrm{b}}/\varepsilon_{\mathrm{K}} \rfloor$. □

With the greedy policy $\pi_t$ from $\hat{Q}_t$, the implemented algorithm's policy is formed by $A_t = \{\pi_k\}_{k=t}^{\infty}$. Next, we discover the relationship between $J^{A_t}$ and $\hat{Q}_t$.

*Proof (Lemma 5)* We assume the current time is $t_0$ and the state is $s_{t_0}$. Considering the implementation of $A_{t_0}$ on the agent for the next $T_{\varepsilon_{\mathrm{H}}}$ steps, one of the following two cases must happen:

(1) There exists at least one time $t \in [t_0, t_0 + T_{\varepsilon_{\mathrm{H}}} - 1]$ that the pair $(s_t, a_t)$ is unknown in $D_t$;

(2) For every time $t \in [t_0, t_0 + T_{\varepsilon_{\mathrm{H}}} - 1]$, $(s_t, a_t)$ is known.

If case (1) occurs, we know that an escape event $E_{t_0}(s_{t_0})$ happens. Then, we have $\max_a \hat{Q}_{t_0}(s_{t_0}, a) - J^{A_{t_0}}(s_{t_0}) \le \mathbb{I}\{E_{t_0}(s_{t_0})\} V_{\mathrm{b}}$, based on the definition of the maximum difference of returns.

For case (2), during the time interval $[t_0, t_0 + T_{\varepsilon_{\mathrm{H}}} - 1]$, the pair $(s_t, a_t)$ is always known in $D_t$. That means $D_t$, $\hat{Q}_t$, $A_t$ remain unchanged for $T_{\varepsilon_{\mathrm{H}}}$ steps starting from $t_0$. For simplicity, let $t_0 = 0$, $D$ for $D_{t_0}$, $\hat{Q}$ for $\hat{Q}_{t_0}$ and $\pi$ for $\pi_{t_0}$. So, we can write the history of agent's states and actions as $(s_0, a_0, s_1, a_1, \ldots, s_{T_{\varepsilon_{\mathrm{H}}}-1}, a_{T_{\varepsilon_{\mathrm{H}}}-1})$. First, we consider $s_0$ at the beginning. Because $\pi$ is greedy of $\hat{Q}$, we have $\max_a \hat{Q}(s_0, a) = \hat{Q}(s_0, a_0)$. As no escape event happens, $C(s_0, a_0)$ is not empty and there exists a sample $(\hat{s}_0, a_0, \hat{r}_0, \hat{s}'_0) \in C(s_0, a_0)$ such that $\hat{s}'_0$ and $s_1$ have $\left| \max_{a'_1} \hat{Q}(\hat{s}'_0, a'_1) - \max_{a'_2} \hat{Q}(s_1, a'_2) \right| \le \varepsilon_{\mathrm{K}}$, by the definition of known. Combined with the definition of $\hat{Q}$, we have

$$\begin{aligned}
\hat{Q}(s_0, a_0) &= \min_{\hat{s}_i \in C(s_0, a_0)} \left[ \hat{r}_i + \gamma \max_{a'} \hat{Q}(\hat{s}'_i, a') \right] \\
&\le \hat{r}_0 + \gamma \max_{a'} \hat{Q}(\hat{s}'_0, a') \\
&= \hat{r}_0 + \gamma\varepsilon_{\mathrm{K}} + \gamma \max_{a'} \hat{Q}(s_1, a').
\end{aligned}$$

Meanwhile, according to the Bellman principle,

$J^{\pi}(s_0) = r_0 + \gamma J^{\pi}(s_1).$

In this way, we can derive that

$$\begin{aligned}\max_a \hat{Q}(s_0,a) - J^{\pi}(s_0) &\\ \le \hat{r}_0 + \gamma\varepsilon_{\mathrm{K}} + \gamma \max_{a'} \hat{Q}(s_1,a') - r_0 - \gamma J^{\pi}(s_1) &\\ \le \alpha\delta + \gamma\varepsilon_{\mathrm{K}} + \gamma\left[\max_{a'} \hat{Q}(s_1,a') - J^{\pi}(s_1)\right]. &\end{aligned}$$

For $\max_{a'} \hat{Q}(s_1,a') - J^{\pi}(s_1)$, the analysis is the same. So, by iterative inferences, we have

$$\max_a \hat{Q}(s_0,a) - J^{\pi}(s_0) \le \sum_{t=0}^{T_{\varepsilon_{\mathrm{H}}}-1} \gamma^t(\alpha\delta + \gamma\varepsilon_{\mathrm{K}}) + \gamma^{T_{\varepsilon_{\mathrm{H}}}} V_{\mathrm{b}}.$$

By the definition of $T_{\varepsilon_{\mathrm{H}}}$, we know $\gamma^{T_{\varepsilon_{\mathrm{H}}}} V_{\mathrm{b}} \le \varepsilon_{\mathrm{H}}$. Then,

$$\begin{aligned}\max_a \hat{Q}(s_0,a) - J^{\pi}(s_0) &\le \sum_{t=0}^{T_{\varepsilon_{\mathrm{H}}}-1} \gamma^t(\alpha\delta + \gamma\varepsilon_{\mathrm{K}}) + \varepsilon_{\mathrm{H}}\\ &\le \sum_{t=0}^{\infty} \gamma^t(\alpha\delta + \gamma\varepsilon_{\mathrm{K}}) + \varepsilon_{\mathrm{H}}\\ &\le \frac{\alpha\delta}{1-\gamma} + \frac{\gamma}{1-\gamma}\varepsilon_{\mathrm{K}} + \varepsilon_{\mathrm{H}}.\end{aligned}$$

Summing up case (1) and (2), we can conclude that the following inequation is satisfied for every time

$$\begin{aligned}&\max_a \hat{Q}_t(s_t,a) - J^{A_t}(s_t)\\ &\le \mathbb{I}\{E_t(s_t)\}V_{\mathrm{b}} + \frac{\alpha\delta}{1-\gamma} + \frac{\gamma}{1-\gamma}\varepsilon_{\mathrm{K}} + \varepsilon_{\mathrm{H}}\end{aligned}$$

□

In the end, we give the proofs of our main theorems in kd-CPAC.

*Proof (Theorem 1)* From Lemma 4, for any $s$ and $a$, their $C_t(s,a)$ has maximum $N_{\mathrm{c}}$ samples stored in it. In other words, at any action, an arbitrary state is surrounded by at most $N_{\mathrm{c}}$ $\delta$-close neighboring samples. Based on Definition 6 of largest minimal cover $N_{\delta}$ and the lemma (Lemma 4.5, Exploration Bound) in Kakade et al. [21], for $|A|$ actions, at most $N_{\delta}N_{\mathrm{c}}|A|$ samples can be stored in data set. Considering the worst case, each sample corresponds to an escape event with $T_{\varepsilon_{\mathrm{H}}}$ length. So, in the whole process of kd-CPAC, the total number of steps when escape event happens is bounded by $N_{\delta}N_{\mathrm{c}}|A|T_{\varepsilon_{\mathrm{H}}}$.

Meanwhile, combine Lemma 3 and Lemma 5 and let $\varepsilon = \frac{2\alpha\delta+\gamma\bar{\omega}(\beta\delta)}{1-\gamma} + \frac{\gamma}{1-\gamma}\varepsilon_{\mathrm{K}} + \varepsilon_{\mathrm{H}}$, we have $J^{A_t}(s_t) \ge \max_a Q(s,a) - \varepsilon - \mathbb{I}\{E_t(s_t)\}V_{\mathrm{b}}$. So, at one moment, if escape event is not encountered, the greedy action of kd-CPAC is near-optimal. Then, it is inferred that the sum of steps when implemented policies are non-optimal is no more than the number of steps when escape event happens, namely

$$\mathrm{PMC}(\varepsilon) \le \sum_{t=0}^{\infty}\{E_t(s_t)\} \le N_{\delta}N_C|A|\log_{1/\gamma}\frac{V_{\mathrm{b}}}{\varepsilon_{\mathrm{H}}}$$

The proof is completed. □

*Proof (Theorem 2)* Based on the supposition of the online problem, during the implementation of kd-CPAC, if no escape event happens in one episode, it is indicated that in the following episodes, trajectories are all the same and no more samples are stored in data set. Then, the algorithm stops and outputs a near-optimal policy from the result of Theorem 1.

From the above analysis, the maximum number of stored samples is $N_{\delta}N_{\mathrm{c}}|A|$. Considering the worst case, the upper bound of episodes that encounter escape events is $N_{\delta}N_{\mathrm{c}}|A|$. So, after this maximum number of episodes, kd-CPAC is determinate to stop and output a near-optimal policy. In other words, the upper bound of running time is $N_{\delta}N_{\mathrm{c}}|A|T_{\mathrm{episode}}$ steps. □

## 6 Examples

In this section, we apply kd-CPAC to three different problems, mountain car, inverted pendulum, and cart–pole balancing. Mountain car is a two-dimensional system with failure and success terminals. Inverted pendulum is also two-dimensional but without terminals. And cart–pole balancing problem has four state variables, and it has only a failure terminal but no success one.

In the implementation, a distance metric $d$ is required. Here, we choose $d$ in a modified version of maximum norm. For two states $s_1$ and $s_2$, their distance is defined by

$$d(s_1,s_2) = \max_j \left|\frac{s_1^j - s_2^j}{S_{\mathrm{sup}}^j - S_{\mathrm{inf}}^j}\right|$$

where $S_{\mathrm{sup}}$ and $S_{\mathrm{inf}}$ are the upper and lower bound of the state space, and the superscript $j$ indicates the $j$-th dimension. The value is normalized by $S_{\mathrm{sup}}$ and $S_{\mathrm{inf}}$, which is based on the same consideration in the calculation of variances when choosing split dimension in kd-tree.

### 6.1 Mountain car

The mountain car is a widely used system to test RL algorithms [24, 31]. A schematic is illustrated in Fig. 1. At the beginning, the car is initialized at the bottom position ($p = -0.5$). By applying a horizontal force, the car can move left and right. The target is to reach the top of the mountain ($p = 1$). The system dynamics is denoted by

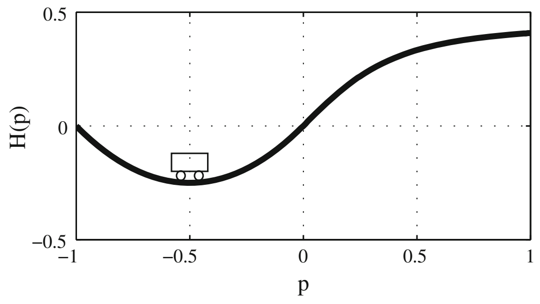

**Fig. 1** Schematic of mountain car system

$$\ddot{p} = \frac{1}{1 + \left(\frac{\mathrm{d}H(p)}{\mathrm{d}p}\right)^2}\left(u - g\frac{\mathrm{d}H(p)}{\mathrm{d}p} - \dot{p}^2\frac{\mathrm{d}H(p)}{\mathrm{d}p}\frac{\mathrm{d}^2H(p)}{\mathrm{d}^2p}\right)$$

where $p \in [-1, 1]$ m is the horizontal position of the car, $\dot{p} \in [-3, 3]$ m/s is its velocity, $u \in [-4, 4]$ N is the horizontal force, $g = 9.81$ m/s$^2$ is the gravitational acceleration, and $H$ denotes the shape of the hill, defined as

$$H(p) = \begin{cases} p^2 + p, & \text{if } p < 0 \\ \frac{p}{\sqrt{1 + 5p^2}}, & \text{if } p \geq 0 \end{cases}.$$

In the simulation, the state variable is $s = [p, \dot{p}]^{\mathrm{T}}$ and the action set is discretized by $A = \{-4,4\}$. Whenever the car passes the left edge ($p = -1$) or its velocity is over three ($|\dot{p}| > 3$), we regard it as a failure and stop the driving. The success condition is the car reaching the right edge ($p = 1$) within the speed limit ($|\dot{p}| \leq 3$). So, the failure terminal is assigned with a low value, $V(s_{\text{failure}}) = -100$, while the success one with $V(s_{\text{success}}) = 0$. In the middle of the process, each step has a reward $r = -1$. The discounted factor is $\gamma = 0.95$, and the sample time is 0.1 s. The episode length $T_{\text{episode}}$ is set to 10 s and the initial state $s_0 = [-0.5, 0]^{\mathrm{T}}$. The split condition $N_{\text{split}}$ chooses 20.

First, we fix the value of neighboring distance δ as 0.01 and study the impact of different known errors $\varepsilon_{\mathrm{H}}$ on the final learned policies of kd-CPAC. To evaluate a policy, the time length in an episode to success is adopted as its performance. The results are illustrated in Table 1. Viewed from the tendency, it is obvious that smaller known errors lead to more optimal policies. This is consistent with our theoretical results. Furthermore, if the known error is too large, the policy can fail to move the car to the goal like the last experiment in Table 1.

Then, we fix known error $\varepsilon_{\mathrm{H}}$ to 1.0 and examine the influence of neighboring distance $\delta$. Similarly, we can conclude from Table 2 that a large neighboring distance leads to a bad policy. These two groups of experiments are agreed with our theoretical results that the smaller values $\delta$ and $\varepsilon_{\mathrm{H}}$ choose, the more optimal policy kd-CPAC learns.

**Table 1** Mountain car: learned policies' performance of kd-CPAC at different known errors when $\delta = 0.01$

| $\varepsilon_{\mathrm{H}}$ | 1.0 | 3.0 | 5.0 | 7.0 | 9.0 |
|---|---|---|---|---|---|
| Length to success(s) | 1.9 | 1.9 | 4.0 | 6.2 | Fail |

**Table 2** Mountain Car: Learned policies' performance of kd-CPAC at different neighboring distances when $\varepsilon_{\mathrm{H}} = 1.0$

| $\delta$ | 0.01 | 0.015 | 0.02 | 0.025 | 0.03 |
|---|---|---|---|---|---|
| Length to success(s) | 1.9 | 1.9 | 1.9 | 2.0 | Fail |

Next, let $\delta = 0.02$ and $\varepsilon_{\mathrm{H}} = 1.0$ and observe the process of kd-CPAC. After 100 trials of running, the algorithm stops and a total of 1,216 samples are stored. The stored samples at each action are presented in Fig. 2, combined with the partitions of state space by the leaves in kd-trees. These figures illustrate that kd-tree can efficiently store samples for our algorithm. Apply the learned policy to the system starting from the initial state, the trajectories are depicted in Fig. 3. It is revealed that after 1.9 s, the car successfully reaches the goal. Besides, the policy is so efficient that only one turn of actions in the episode leads to the success.

## 6.2 Inverted pendulum

In the second simulation, we adopt the inverted pendulum. It is a common example to estimate online algorithms [2].The inverted pendulum is a device that rotates a mass in a vertical plane and is driven by a DC motor. A schematic is presented in Fig. 4, and its dynamics can be denoted by

$$\ddot{\alpha} = \frac{1}{J}\left(mgl\sin(\alpha) - b\dot{\alpha} - \frac{K^2}{R}\dot{\alpha} + \frac{K}{R}u\right)$$

where $\alpha$ and $\dot{\alpha}$ are the angle and angular velocity of the pendulum, satisfying the bound $[-\pi,\ \pi]$ rad and $[-15\pi,\ 15\pi]$ rad/s, respectively. $u$ is the control action applied to the DC motor and constrained to $[-3,\ 3]$V. For simulation, dynamics parameters are adopted the same as [2], given in Table 3, and the sample time is set to 0.01 s.

The goal is to swing up the pendulum from the bottom and balance it at the top. So, the state input is $s = [\alpha, \dot{\alpha}]^{\mathrm{T}}$ and the control action is discretized into three discrete values, $A = \{-3,\ 0,\ 3\}$. The reward is designed by $r(s,a) = -s^T Qs$, where $Q = \mathrm{diag}(5,\ 0.1)$. The discounted factor is set $\gamma = 0.98$. The episode length $T_{\text{episode}}$ is 6 s and each trial starts from $[\pi, 0]^{\mathrm{T}}$.

In this experiment, we still choose $N_{\text{Split}} = 20$ but set $\delta = 0.005$ and $\varepsilon_{\mathrm{K}} = 30.0$. After 143 episodes of learning, the algorithm stops and it stores 29,684 samples. Stored

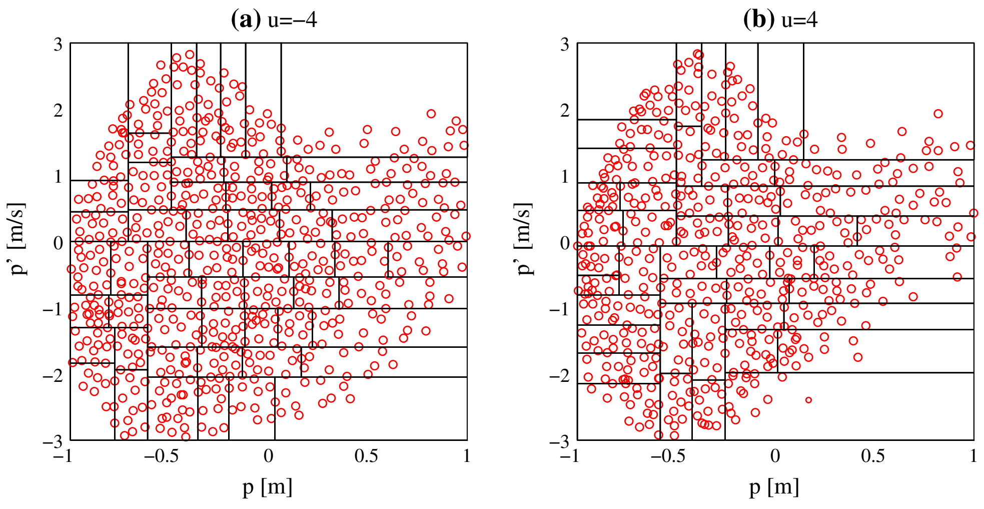

**Fig. 2** Mountain car: partitions of state space and the stored samples at each action of kd-trees with $\delta = 0.02$ and $\varepsilon_H = 1.0$

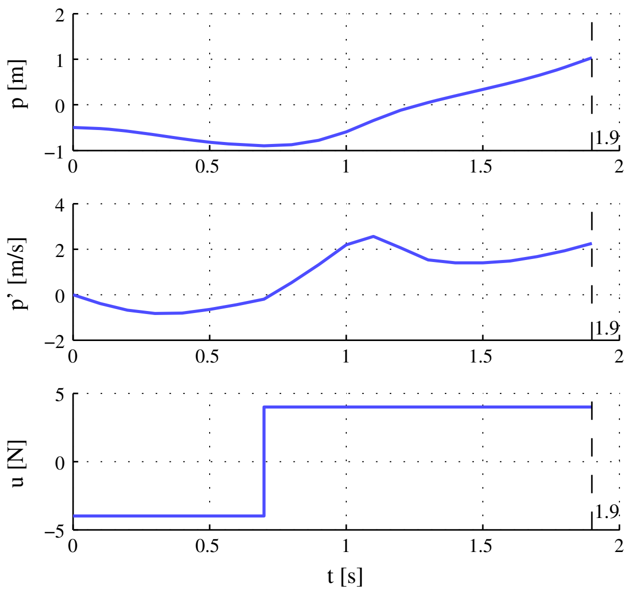

**Fig. 3** Mountain car: trajectories of states and actions under the learned policy of kd-CPAC with $\delta = 0.02$ and $\varepsilon_H = 1.0$

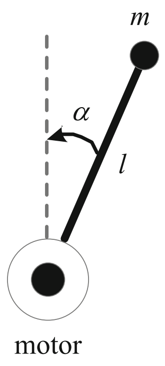

**Fig. 4** Schematic of the inverted pendulum

samples in three kd-trees are depicted in Fig. 5. As the problem is complicated, more samples are stored and the state space is partitioned by kd-trees to smaller sizes. Still, space is split based on the samples' distribution in a high efficient way of utilization.

**Table 3** Parameters of inverted pendulum

| Symbol | Value | Meaning |
|---|---|---|
| $m$ | 0.055 | Mass |
| $g$ | 9.81 | Gravitational acceleration |
| $l$ | 0.042 | Distance from center to mass |
| $J$ | 1.91e−4 | Moment of inertia |
| $b$ | 3e−6 | Viscous damping |
| $K$ | 0.0536 | Torque constant |
| $R$ | 9.5 | Rotor resistance |

Next, the learned policy is applied to the system to observe its performance. For comparison, an offline model-based Fuzzy Q-Iteration from [2] is also applied to the same system. In its implementation, we set triangular fuzzy partitions with 51 equidistant cores for both state variables. After the offline learning, a convergent policy is obtained. We apply the two policies learned by kd-CPAC and Fuzzy Q-Iteration to inverted pendulum, and their results are presented in Fig. 6. It is revealed that the policy of kd-CPAC (blue solid lines) takes less steps to swing up and balance the pendulum than Fuzzy Q-Iteration (green dashed lines). So, it is indicated that our algorithm can derive a more optimal policy than Fuzzy Q-Iteration, even though it is online and has no information about the system, while Fuzzy Q-Iteration is offline. The reason can be explained by the following analysis. Observing the chosen actions of Fuzzy Q-Iteration from the beginning to about 0.75 s, the policy almost selects the same action. This selection at first is good to push the pendulum, but when the pendulum cannot be pushed further more (at 0.4 s), the old action is in fact a resistance for the movement. But in kd-CPAC, actions are changed earlier and more frequently

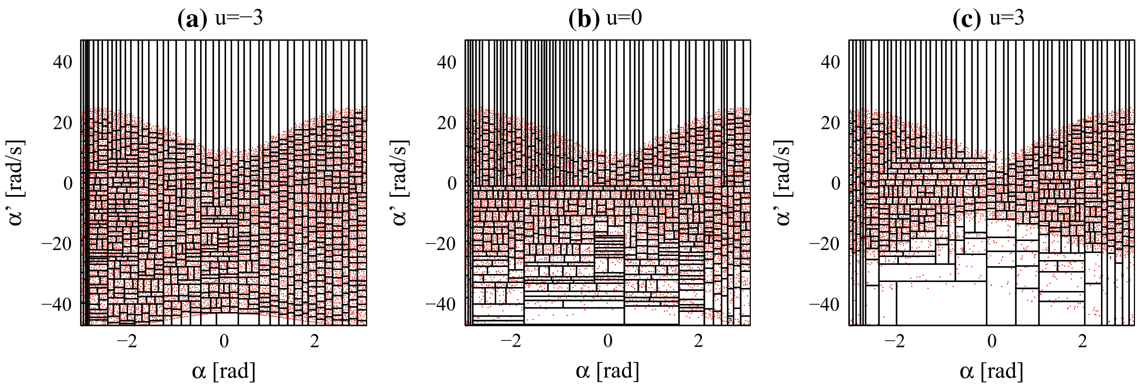

**Fig. 5** Inverted pendulum: partitions of state space and the stored samples at each action of kd-trees with $\delta = 0.005$ and $\varepsilon_K = 30.0$

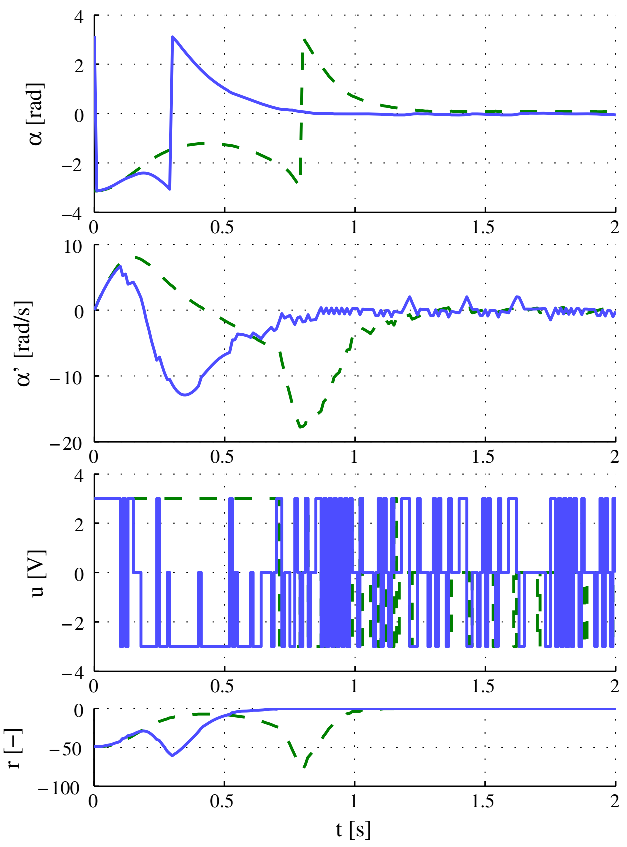

**Fig. 6** Inverted pendulum: trajectories of states, actions, and rewards. The blue solid lines indicate the policy of kd-CPAC, while the green dashed lines refer to the policy of Fuzzy Q-Iteration with 51 × 51 triangular fuzzy partitions

to adjust for the movement of the pendulum. In this way, the pendulum in kd-CPAC is swung up more quickly.

In [23], Bernstein and Shimkin apply their ARL algorithm, which is also a PAC algorithm, to the same problem. However, their implementation relies on some system parameters which is commonly unknown in practice. While in our algorithm, no system information is required and this benefits a wider application for our algorithm compared to them.

## 6.3 Cart–pole balancing problem

The last example is a cart–pole balancing problem. The task is to balance a pole upwards, which is hinged on a cart (as illustrated in Fig. 7). The system has four state variables: the position of the cart $x$, the pole angle with respect to the vertical axis $\theta$, and their derivatives $(\dot{x}, \dot{\theta})$. A force $u$ is applied to the cart in the $x$-direction to control the system. The dynamics can be denoted by

$$\begin{bmatrix} lm_p\cos\theta & -(m_c+m_p) \\ \frac{4}{3}l & -\cos\theta \end{bmatrix}\begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} lm_p\dot{\theta}^2\sin\theta+\mu_c sign(\dot{x}) \\ g\sin\theta - \dfrac{\mu_p\dot{\theta}}{lm_p} \end{bmatrix} + \begin{bmatrix} u \\ 0 \end{bmatrix}$$

where parameters are adopted the same values as in [32], listed in Table 4.

In our experiment, the state vector is $s = [x, \dot{x}, \theta, \dot{\theta}]^{\mathrm{T}}$ and the control actions include 2 values, $A = \{-5,\ 5\}$. The failure condition is whenever the cart drives outside the track ($|x| > 2.4$) or the pole drops over 20° ($|\theta| > \frac{20}{180}\pi$). Besides, we support the derivatives are bounded by $|\dot{x}| \le 8$ and $|\dot{\theta}| \le \pi$. Considering the goal is to balance the pole around the vertical position and drive the cart near $x = 0$, the reward function is defined considering the difference between two adjacent states. Given $[x_k, \dot{x}_k, \theta_k, \dot{\theta}_k]^{\mathrm{T}}$ and $[x_{k+1}, \dot{x}_{k+1}, \theta_{k+1}, \dot{\theta}_{k+1}]^{\mathrm{T}}$, the reward at the $k$-th step is denoted by

$$r_k = 100|\theta_k - \theta_{k+1}| + 5|x_k - x_{k+1}|.$$

Actions that make states closer to the vertical and zero-point position gain more rewards. Moreover, the value for failure is assigned by $V(s_{\text{failure}} = 1{,}000)$. The length of an episode is set to 60 s, and the sample time is 0.05 s. The discounted factor is $\gamma = 0.95$. The starting state is $[0, 0, \frac{10}{180}\pi, 0]^{\mathrm{T}}$.

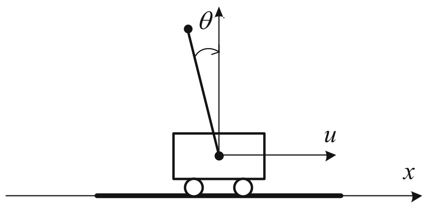

**Fig. 7** Schematic of the cart–pole system

**Table 4** Parameters of cart–pole

| Symbol | Value | Meaning |
|---|---|---|
| $u_{\max}$ | 5 | Maximal force |
| $g$ | 9.81 | Gravity acceleration |
| $m_p$ | 0.5 | Mass of the pole |
| $m_c$ | 1.0 | Mass of the cart |
| $l$ | 0.5 | Length of the pole |
| $\mu_c$ | 1.0 | Coefficient of friction of cart on track |
| $\mu_p$ | 0.1 | Coefficient of friction of pivot |
| $L$ | 2.4 | Half length of the track |

However, the learning of cart–pole balancing problem is difficult. On the one hand, the states include four variables, while the previous two systems include only two. It is more complicated to control four variables. Besides, it is much easier for the pole to loss balance in the system. It can fall down after only few steps from the beginning, which makes the learning process has to restart again and again frequently. In this way, the implementation is much longer with more episodes to stop and output a satisfying policy.

In kd-CPAC, split condition $N_{\text{split}}$ is still equal to 20. The neighboring distance $\delta$ is set to 0.02, and the known error $e_K$ is 100. After learning, the final policy is applied to the system from the starting state and the system can be controlled within the limit of $\theta$ and $x$ successfully. The corresponding trajectories are depicted in Fig. 8. To display clearly, only the first 20 s are presented in the figure. It is revealed that the cart and the pole are conducted toward zero point from the initial deflected position after few seconds. So, the learned policy from kd-CPAC can successfully balance the system around zero-point area. This simulation illustrates that our algorithm can perform well even for complicated systems.

## 7 Conclusion

In this paper, we consider continuous deterministic systems and propose a new online RL algorithm, kd-CPAC. During the online running, the algorithm selectively stores samples and utilizes them directly to produce policies. These policies are prone to explore unvisited areas, which helps to collect system information. By rigorous theoretical analysis, we prove the algorithm satisfies the PAC principle. During the whole process, the sum of steps when algorithm implements non-optimal actions is finite and bounded. Furthermore, for online cases, the running time for kd-CPAC to produce a near-optimal policy also has an upper bound.

The near optimality and the finite time bound are the main advantages of our algorithm compared to conventional online RL algorithms. These conventional algorithms consider less about the optimality of their final learned policies and the running time of their implementation. Besides, our algorithm is totally model-free. It directly utilizes the online data, and no system parameters are required.

To avoid the dependence on approximation structures in the implementation, we utilize the online data directly. It benefits the algorithm with high efficient utilization of online data. To store samples, a kd-tree technique is used. It helps to divide the state space according to samples and store them in a tree structure. Based on kd-tree, it is convenient for the algorithm to locate samples and search for neighboring samples for arbitrary states.

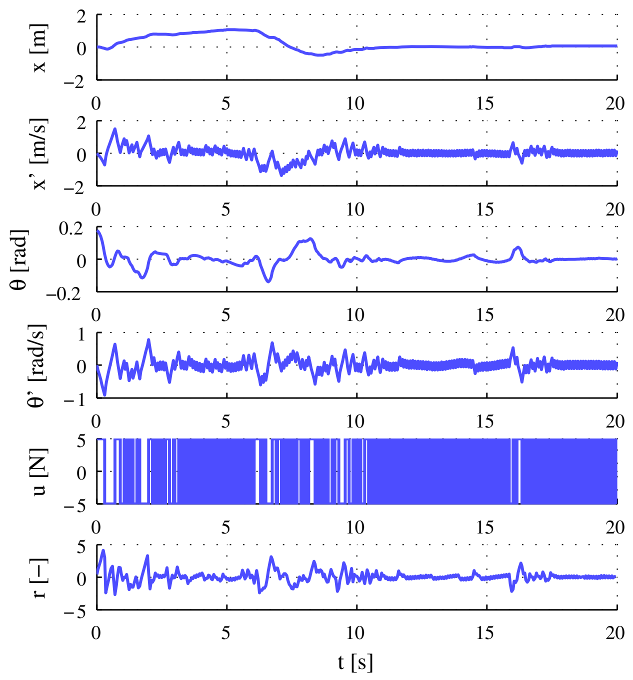

**Fig. 8** Cart–pole balancing: trajectories of states and actions under the learned policy of kd-CPAC with $\delta = 0.02$ and $\varepsilon_H = 100$

**Acknowledgments** This work is supported by National Natural Science Foundation of China (NSFC) under Grants No. 61273136, No. 61034002, and Beijing Natural Science Foundation under Grant No. 4122083.

## Appendix

Based on value iteration, to calculate $\hat{Q}_t$, we first initialize a function $\hat{Q}_t^{(0)}$ which can be assigned to any value. Usually, $\hat{Q}_t^{(0)}$ is equal to 0 or $V_{\max}$. Then, calculate the $Q$ values of stored samples $(\hat{s}, \hat{a}, \hat{r}, \hat{s}') \in D_t$ by

$$\hat{q}_t^{(0)}(\hat{s}, \hat{a}) = \hat{r} + \gamma \max_{a'} \hat{Q}_t^{(0)}(\hat{s}', a').$$

Furthermore, a new $\hat{Q}_t^{(1)}$ can be obtained by

$$\hat{Q}_t^{(1)}(s,a) = \begin{cases} \min\limits_{\hat{s}_i \in C_t(s,a)} \hat{q}_t^{(0)}(\hat{s}_i, a), & \text{if } C_t(s,a) \neq \emptyset \\ V_{\max}, & \text{otherwise} \end{cases}$$

The above equation is totally equal to the process of calculating $\hat{Q}_t^{(1)}$ from $\hat{Q}_t^{(0)}$ by (5). Then, this calculation is iterated.

In conclusion, suppose we have $\hat{Q}_t^{(j)}$ of the $j$-th iteration, calculate $Q$ values of stored samples using

$$\hat{q}_t^{(j)}(\hat{s}, \hat{a}) = \hat{r} + \gamma \max_{a'} \hat{Q}_t^{(j)}(\hat{s}', a').$$

Then, $\hat{Q}_t^{(j+1)}$ at the $(j+1)$-th iteration is obtained by

$$\hat{Q}_t^{(j+1)}(s,a) = \begin{cases} \min\limits_{\hat{s}_i \in C_t(s,a)} \hat{q}_t^{(j)}(\hat{s}_i, a), & \text{if } C_t(s,a) \neq \emptyset \\ V_{\max}, & \text{otherwise} \end{cases}$$

As the above process is a variant of solving (5) by value iteration, so it is convergent and the result is the same with directly calculating $\hat{Q}_t$ by value iteration. Moreover, in the process, the only need is storing $Q$ values of samples, and the values of $\hat{Q}_t$ over the whole state space are easy to obtain.

## References

1. Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. MIT Press, Cambridge
2. Busoniu L, Babuska R, De Schutter B, Ernst D (2010) Reinforcement learning and dynamic programming using function approximators. CRC Press, New York
3. Tan AH, Ong YS, Tapanuj A (2011) A hybrid agent architecture integrating desire, intention and reinforcement learning. Expert Syst Appl 38(7):8477–8487
4. Tang L, Liu Y-J, Tong S (2014) Adaptive neural control using reinforcement learning for a class of robot manipulator. Neural Comput Appl 25(1):135–141
5. Wang D, Liu D, Zhao D, Huang Y, Zhang D (2013) A neural-network-based iterative GDHP approach for solving a class of nonlinear optimal control problems with control constraints. Neural Comput Appl 22(2):219–227
6. Wei Q, Liu D (2014) Stable iterative adaptive dynamic programming algorithm with approximation errors for discrete-time nonlinear systems. Neural Comput Appl 24(6):1355–1367
7. Wang B, Zhao D, Alippi C, Liu D (2014) Dual heuristic dynamic programming for nonlinear discrete-time uncertain systems with state delay. Neurocomputing 134:222–229
8. Watkins C (1989) Learning from delayed rewards. PhD thesis, Cambridge University, Cambridge
9. ten Hagen S, Kröse B (2003) Neural Q-learning. Neural Comput Appl 12(2):81–88
10. Rummery GA, Niranjan M (1994) On-line Q-learning using connectionist systems. Tech. Rep. TR 166, Cambridge University Engineering Department, Cambridge, England
11. Liu D, Wang D, Zhao D, Wei Q, Jin N (2012) Neural-network-based optimal control for a class of unknown discrete-time nonlinear systems using globalized dual heuristic programming. IEEE Trans Autom Sci Eng 9(3):628–634
12. Thrun SB (1992) The role of exploration in learning control. In: White D, Sofge D (eds) Handbook for intelligent control: neural, fuzzy and adaptive approaches. Van Nostrand Reinhold, Florence, Kentucky 41022
13. Zhao D, Hu Z, Xia Z, Alippi C, Wang D (2014) Full range adaptive cruise control based on supervised adaptive dynamic programming. Neurocomputing 125:57–67
14. Zhao D, Wang B, Liu D (2013) A supervised actor-critic approach for adaptive cruise control. Soft Comput 17(11):2089–2099
15. Zhao D, Bai X, Wang F, Xu J, Yu W (2011) DHP for coordinated freeway ramp metering. IEEE Trans Intell Transp Syst 12(4):990–999
16. Bai X, Zhao D, Yi J (2009) The application of ADHDP($\lambda$) method to coordinated multiple ramps metering. Int J Innov Comput 5(10(B)):3471–3481
17. Kearns M, Singh S (2002) Near-optimal reinforcement learning in polynomial time. Mach Learn 49(2–3):209–232
18. Brafman RI, Tennenholtz M (2003) R-max—a general polynomial time algorithm for near-optimal reinforcement learning. J Mach Learn Res 3:213–231
19. Strehl AL, Littman ML (2005) A theoretical analysis of model-based interval estimation. In: Proceedings of 22nd international conference on machine learning (ICML'05), pp 856–863
20. Strehl AL, Li L, Wiewiora E, Langford J, Littman ML (2006) PAC model-free reinforcement learning. In: Proceedings of 23rd international conference on machine learning (ICML'06), pp 881–888
21. Kakade S, Kearns MJ, Langford J (2003) Exploration in metric state spaces. In: Proceedings of 20th international conference on machine learning (ICML'03), pp 306–312
22. Pazis J, Parr R (2013) PAC optimal exploration in continuous space markov decision processes. In: AAAI conference on artificial intelligence
23. Bernstein A, Shimkin N (2010) Adaptive-resolution reinforcement learning with polynomial exploration in deterministic domains. Mach Learn 81(3):359–397
24. Munos R, Moore A (2002) Variable resolution discretization in optimal control. Mach Learn 49(2–3):291–323
25. Ernst D, Geurts P, Wehenkel L (2005) Tree-based batch mode reinforcement learning. J Mach Learn Res 6:503–556
26. Preparata FP, Shamos MI (1985) Computational geometry: an introduction. Springer, Berlin
27. Li H, Liu D (2012) Optimal control for discrete-time affine nonlinear systems using general value iteration. IET Control Theory Appl 6(18):2725–2736
28. Al-Tamimi A, Lewis FL, Abu-Khalaf M (2008) Discrete-time nonlinear HJB solution using approximate dynamic programming: convergence proof. Trans Syst Man Cyber Part B 38(4):943–949
29. Liu D, Yang X, Li H (2013) Adaptive optimal control for a class of continuous-time affine nonlinear systems with unknown internal dynamics. Neural Comput Appl 23(7–8):1843–1850

30. Zuo L, Xu X, Liu C, Huang Z (2013) A hierarchical reinforcement learning approach for optimal path tracking of wheeled mobile robots. Neural Comput Appl 23(7–8):1873–1883
31. Schoknecht R, Riedmiller M (2003) Reinforcement learning on explicitly specified time scales. Neural Comput Appl 12(2):61–80
32. Neumann G (2005) The reinforcement learning toolbox: reinforcement learning for optimal control tasks. Master's thesis, Technischen Universität (University of Technology) Graz