Mach Learn (2017) 106:1569–1598
DOI 10.1007/s10994-017-5650-8

Offline reinforcement learning with task hierarchies

Devin Schwab[1] · **Soumya Ray**[1]

Received: 30 March 2016 / Accepted: 20 June 2017 / Published online: 12 July 2017
© The Author(s) 2017

Abstract In this work, we build upon the observation that offline reinforcement learning (RL) is synergistic with task hierarchies that decompose large Markov decision processes (MDPs). Task hierarchies can allow more efficient sample collection from large MDPs, while offline algorithms can learn better policies than the so-called "recursively optimal" or even hierarchically optimal policies learned by standard hierarchical RL algorithms. To enable this synergy, we study sample collection strategies for offline RL that are consistent with a provided task hierarchy while still providing good exploration of the state-action space. We show that naïve extensions of uniform random sampling do not work well in this case and design a strategy that has provably good convergence properties. We also augment the initial set of samples using additional information from the task hierarchy, such as state abstraction. We use the augmented set of samples to learn a policy offline. Given a capable offline RL algorithm, this policy is then guaranteed to have a value greater than or equal to the value of the hierarchically optimal policy. We evaluate our approach on several domains and show that samples generated using a task hierarchy with a suitable strategy allow significantly more sample-efficient convergence than standard offline RL. Further, our approach also shows more sample-efficient convergence to policies with value greater than or equal to hierarchically optimal policies found through an online hierarchical RL approach.

Keywords Reinforcement learning · Hierarchical reinforcement learning · MaxQ · Least-squares policy iteration (LSPI)

Editors: Thomas Gärtner, Mirco Nanni, Andrea Passerini, and Celine Robardet.

✉ Devin Schwab
dts34@case.edu

Soumya Ray
sray@case.edu

[1] Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106, USA

1 Introduction and motivation

Reinforcement learning (RL) approaches are a standard paradigm for solving sequential decision making problems without prior knowledge of the world. Through interaction with the world, over time, an RL agent builds a policy that maps world states to actions that maximize some measure of the agent's expected utility. This paradigm has been used successfully in many practical applications from job shop scheduling (Zhang and Dietterich 1995) to helicopter control (Abbeel et al. 2010). More recently, these techniques have been successfully applied to complex games such as Atari (Mnih et al. 2015) and Go (Silver et al. 2016).

RL algorithms can be broadly divided into online and offline approaches. Online RL algorithms, such as Q-learning (Watkins and Dayan 1992), interleave interacting with the domain and learning a policy, while offline approaches separate the interaction/sample collection phase and learning phases. Each has advantages and disadvantages. Online approaches can often be computationally efficient and adapt quickly to new observations. Offline approaches, such as Least-Squares Policy Iteration (LSPI) (Lagoudakis and Parr 2003) on the other hand, can easily take advantage of prior collections of samples, and generally provide robust convergence guarantees in the presence of approximation and a finite number of samples (Lazaric et al. 2012). In our work, we focus on offline RL approaches.

In our work, we build upon the observation that task hierarchies are synergistic with offline RL approaches. In particular, a key step in offline RL is the interaction/sample collection step. Since the learned policy is based off these samples, to ensure that a good policy is learned, it is important to collect a representative set of samples from the environment. Often this is done through uniform random sampling, so that each state and action will be eventually sampled often enough. However, building such a set of samples can be challenging in large MDPs. For one thing, some actions may be dangerous to the agent in some states. Secondly, the agent may waste time sampling actions in states that are completely irrelevant to the final learned policy. Finally, it may be difficult for an agent using uniform random sampling to collect samples from critical parts of the state space, for example if these regions are behind bottlenecks in the state-action space. We argue that these difficulties can be addressed through the use of a task hierarchy to structure the sampling process. In this way, a task hierarchy can be used to improve offline RL.

On the other hand, standard hierarchical RL approaches can benefit from offline RL as well. One advantage is that offline algorithms often have good convergence guarantees (Wiering and van Otterlo 2012) that may be extendable to the HRL case. A second advantage is that we can sidestep a key difficulty in HRL: how to balance local optimality within a subtask with global optimality of the overall task. For example, in MAXQ (Dietterich 2000), in order to guarantee that the learner produces the best hierarchical policy, "pseudo-rewards" may need to be defined for some subtasks to force the local policy to be sub-optimal. However, the definition of these rewards generally pre-supposes quite detailed knowledge of the optimal policy, resulting in a chicken-and-egg problem. While some work has explored how to learn pseudo-rewards (Schultink et al 2008; Cao and Ray 2012), in this work we show that using offline RL, we can sidestep this issue entirely. In fact, offline RL algorithms can even learn policies with value *better* than the hierarchically optimal policy, though samples are drawn consistent with a task hierarchy, as we show in our experiments.

The key observation we use to integrate task hierarchies with offline RL is that the hierarchies can be used to structure the sampling process for an offline learner. However, we first show that naïve extensions of uniform random sampling do not work in this case. We then develop a strategy that has good convergence properties in terms of the distribution of

samples. We also describe how to augment the set of samples we collect in this way by using additional information in the task hierarchy, such as state abstraction. Finally, we present empirical evidence showing that this approach improves both upon standard offline RL and standard hierarchical RL learners in several domains.

2 Background and related work

Reinforcement Learning (RL) algorithms model the task being solved as a Markov Decision Process (MDP). An MDP is defined by the tuple $(S, S_0, A, P, R, \gamma)$ (Puterman 2005). S is the set of possible states, S_0 the starting state distribution, and A the set of actions available to the agent. P is the transition function which given an (s, a', s) tuple defines the probability of action a transitioning the world from state s to s'. R is the reward function which maps state-action pairs to a real valued number. γ is the discount factor, which specifies the importance of short term rewards versus long term rewards. The agent's goal is to find a policy, $\pi(s) = a$, that maximizes some measure of long-term utility, such as the expected cumulative discounted reward that we use in this paper.

Below, we describe hierarchical reinforcement learning and the Least-Squares Policy Iteration (LSPI) algorithm, the two key components of our work.

2.1 Hierarchical reinforcement learning

Many approaches to HRL have been explored (Barto and Mahadevan 2003). Techniques such as options (Sutton et al. 1999), use fixed, pre-specified sub-policies to decompose a global task. The agent chooses which sub-policy to execute, and then runs the sub-policy to completion before choosing another. Later extensions added the ability to interrupt the options (Comanici and Precup 2010), create a multi-layer hierarchy of options (Stolle and Precup 2002) and automatic learning of option hierarchies (Jonsson 2006). Another major approach is to decompose the global MDP into a hierarchy of semi-MDPs, and learn how to solve each semi-MDP in the hierarchy. Techniques such as Hierarchies of Abstract Machines (Parr and Russell 1998), MAXQ (Dietterich 2000), ALISP (Andre and Russell 2002; Marthi et al. 2005) and H- DQN (Kulkarni et al. 2016) use this type of decomposition as the theoretical base.

Our approach relies on the agent having a task hierarchy that decomposes the overall task. Though our approach can use any decomposition that provides well-defined task-subtask relationships and allows for subtasks to be interrupted (as described in later sections), we use the MAXQ HRL framework (Dietterich 2000) in our experiments. We describe this approach below. MAXQ decomposes a root MDP into a hierarchy of subtasks, each modeled as a Semi-Markov Decision Process (SMDP) with actions that are either other subtasks or primitive actions from the original MDP. Each subtask, M_i, is defined by a tuple (T_i, X_i, A_i, G_i). T_i is the set of terminal states for subtask M_i. M_i may only be called by its parent when the state is not in the set T_i. X_i defines a set of relevant state variables for M_i. A_i is the set of actions (primitive or composite) available within M_i. G_i defines goal states the agent should attempt to reach in M_i. Often all states in T_i and the set of states in G_i are combined into a single termination predicate, E_i. Any (s, a, s') which transitions to a state in E_i is called an exit of the subtask M_i. A pseudo-reward function $\tilde{R}(s, a)$ can be defined over exits to bias the agent towards specific subtask exits. Given such a hierarchy, the value function of a policy at a subtask can be additively decomposed into the value functions of its children in a recursive manner.

A hierarchical policy is a collection of local policies for each subtask in the task hierarchy. The execution of a hierarchical policy involves executing the local policy for each subtask. In standard MAXQ, subtask policies are executed until they or a parent terminate. “Polled execution” is an alternative way of following a hierarchical policy which selects the next action by always starting at the root and recursing down the task hierarchy using each subtask policy until a primitive action is selected and executed.

Hierarchical policies are constrained by the hierarchy and so they cannot guarantee that a globally optimal policy can be found. However, a hierarchical policy can be *hierarchically optimal* if the policy achieves the best reward over all hierarchical policies consistent with a task hierarchy. A hierarchical policy is *recursively optimal* if each local policy for the subtasks is optimal given its childrens’ policies. In the MAXQ framework, such policies are obtained when the pseudo-rewards are uniformly zero. It is well known that recursively optimal policies can be arbitrarily worse than the hierarchically optimal policy, and the hierarchically optimal policy can be arbitrarily worse than the globally optimal policy (Dietterich 2000). Properly chosen pseudo-rewards allow the learner to produce locally suboptimal policies if doing so improves the expected reward of the overall hierarchical policy. On the other hand, misspecifying pseudo rewards can lead to a policy that is worse than the recursively optimal policy or even non-convergence.

2.2 Least-squares policy iteration

In our work, we focus on offline RL algorithms. While any such algorithm can be used with our approach, we build on the Least-Squares Policy Iteration (LSPI) Algorithm (Lagoudakis and Parr 2003). LSPI is an offline, off-policy, approximate policy iteration algorithm. The LSPI pseudo-code is shown in Algorithm 1. LSPI uses a linear combination of basis functions, ϕ, to approximate the optimal Q-function for an MDP. Normally, this type of approximation scheme can have unbounded error, however, the authors prove that formulating the MDP as a system of linear equations formed from the Bellman equations, and then performing a least-squares projection onto the span of ϕ, will lead to a point in the space of ϕ that minimizes the L2-norm to the true Q-function. Generally, using the system of linear equation form of an MDP requires that a model be known, but LSPI provides a model-free way of directly estimating the linear equation matrices A and b from a set of samples D. While LSPI does not explicitly compute a model, it has been shown that it will learn the exact same sequence of value functions as a model-based method due to the replay of the saved samples (van Seijen and Sutton 2015). Lines 7–8 in the algorithm show how to compute the approximate matrix values. If D is a representative distribution of samples then LSPI will converge to a policy with bounded sub-optimality, even in the case of a finite number of samples (Lazaric et al. 2012). In this paper we show how to utilize the task hierarchy to generate the sample set D, so that it contains a representative distribution consistent with the hierarchy.

3 Hierarchical sampling with LSPI

The main contribution of this work is an approach to using a hierarchical decomposition of an MDP as a set of constraints on the sample collection process. These samples can then be used by an offline RL learner, such as LSPI, in order to find a policy for the original, flat MDP. Using this method, we hypothesize that the offline learning algorithm can converge to a policy with a value that is at least as good as the hierarchi-

Algorithm 1 Least-Squares Policy Iteration (LSPI)

1: **procedure** LSPI(D, ϕ, γ, π_0)
2: ▷ D: the set of collected samples (s, a, r, s'), ϕ: Basis functions, γ: Discount factor, π_0: Initial policy
3: $\pi' \leftarrow \pi_0$
4: **repeat**
5: $\pi \leftarrow \pi'$
6: $A, b \leftarrow 0$
7: **for each** $(s, a, r, s') \in D$ **do** ▷ This loop is referred to as LSTDQ in the original LSPI paper
8: $\tilde{A} \leftarrow \tilde{A} + \phi(s, a)\big(\phi(s, a) - \gamma\phi(s', \pi(s'))\big)^\top$
9: $\tilde{b} \leftarrow \tilde{b} + \phi(s, a)r$
10: **end for**
11: $\tilde{w}^\pi \leftarrow \tilde{A}^{-1}\tilde{b}$
12: set π' weights equal to $\tilde{w}^\pi$
13: **until** $\pi \approx \pi'$
14: **return** π
15: **end procedure**

cally optimal policy's value while requiring fewer experience samples than a regular RL method.

The basic sampling algorithm, which we call Offline Hierarchical Reinforcement Learning (OHRL), starts by collecting a set of samples according to the hierarchy. At each state the hierarchy has terminated subtasks and non-terminated subtasks. The terminated subtasks form constraints on the actions that can be taken. If an action is unavailable in every executable subtask, then the hierarchy constrains the agent so that it will not try that action. This forces the agent to spend its time exploring actions that are actually relevant to the particular state. A well-designed hierarchy will disallow actions dangerous to the agent/environment as well as actions not useful to solving the MDP.

The different sampling strategies we describe below differ in how they select actions to be tried (and hence, generate the sampling distribution). As we show below, it is necessary to design the sample collection strategy properly. Otherwise, the collected samples may not allow a good policy to be learned.

We also show that if the hierarchy includes information about actions available at the primitive states and state abstraction, then we are able to use the collected samples together with the hierarchy to derive new samples. These samples are referred to as "derived samples". In Sect. 4 we describe how these are generated and how they enable the learner to take advantage of information in the task hierarchy not explicitly encoded by the samples generated by the strategies above.

The pseudocode in Algorithm 2 summarizes the major steps of our approach as the Offline Hierarchical Reinforcement Learning (OHRL) algorithm. When used with Least-Squares Policy Iteration (LSPI) as the offline policy solver we refer to this algorithm as Hierarchical Sampling Least-Squares Policy Iteration (HS-LSPI).

Algorithm 2 Offline Hierarchical Reinforcement Learning (OHRL)

1: **procedure** OHRL(M, γ)
2: ▷ M : Task Hierarchy, γ : Discount factor
3: $D_{real} \leftarrow SampleHierarchy(M)$ ▷ See section 3 for different sampling strategies.
4: $D_{derived} \leftarrow GenerateDerivedSamples(D_{real}, M, \gamma)$ ▷ Optional. See section 4 for details.
5: $\pi \leftarrow OfflinePolicySolver(D_{real} \cup D_{derived}, \gamma)$ ▷ We use LSPI in this work.
6: **return** π
7: **end procedure**

Before presenting the sample collection strategies, we describe an operation we call "hierarchical projection." This will enable us to theoretically analyze the distribution of the collected samples and compare this distribution to sampling strategies running on the flat MDP.

3.1 Hierarchical projection and the target distribution

Consider an agent executing a hierarchical policy. This could also be viewed as running a *flat* policy on a "hierarchical" MDP, which is defined as follows. Given a flat MDP and a task hierarchy, H, the new hierarchical MDP states are defined by $s_H = (c, s)$ where s is a primitive state and c is a *call stack* containing a path in the hierarchy from the root subtask up to (but not including) a primitive action, a. All of the subtasks in the call stack must be non-terminated for the given state s. Correspondingly, the actions are defined by $A_H = A_C \cup A$, where A are the primitive actions and A_C are the composite actions (subtasks) available in this task hierarchy, plus a special $EXIT$ action. The transition probabilities are defined as follows: $P((c', s')|(c, s), a) = P(s'|s, a)$ if a is a primitive action allowed by the current call stack subtask, s a non-terminal state in the subtask at the top of c and $c' = c$; $P((c', s')|(c, s), a) = 1$ if s is a terminal state of the subtask at the top of c, a is the $EXIT$ action, c' is the stack after this subtask is popped and s' is s, and zero for all other cases when $EXIT$ is called; $P((c', s')|(c, s), a) = 1$ if $a \in A_C$ is an available child for the subtask at the top of c, s is a non-terminal state of a and c' is the stack after a is pushed, and zero for all other cases when actions in A_C are called. Finally, the reward function $R_H((c, s), a_H) = R(s, a)$ for all $((c, s), a_H)$ where $a_H = a$, and zero else.

Given this reduction, we can think about the properties of sampling algorithms in terms of collecting samples using a policy from this equivalent "hierarchical" MDP. How should we collect samples from this MDP? In a normal "flat" MDP, a commonly used strategy is uniform random sampling. In the limit, this procedure will guarantee that every state and action are sampled often enough so the offline learner can converge to the best policy in its hypothesis space. This is usually preferred because, though it may not be the most efficient sampling strategy to find the optimal policy for a particular MDP, it ensures coverage of all state-action pairs, which is necessary to always find the optimal policy in the limit for an arbitrary MDP. When a hierarchy is available, we apply the same principle, but with the *hierarchical MDP*. That is, we would like all non-terminated children actions (both composite and primitive) in each subtask to be chosen with uniform random probability for every state. Again, this will guarantee that in the limit, every state-action pair consistent with the hierarchy will be visited often enough, so that optimal policies can be found for arbitrary MDPs.

A point to note is that though the samples collected in the hierarchical MDP are over the state action space $S_H \times A_H$, we use an offline learner, such as LSPI, that expects samples over $S \times A$. In order to do this, we take the distribution of samples from the hierarchical MDP state-action space and "project" that distribution onto the flat MDP state-action space. In this way the distribution of samples in the flat MDP is biased by the hierarchy structure. We call this operation "hierarchical projection." Note that, because we use a normal, non-hierarchical learning algorithm the final policy we produce is not itself hierarchical, but flat. But through the hierarchical projection operation during sampling, we produce samples (and thus policies) that respect the hierarchical constraints.

Our goal therefore is to design a sampling procedure such that the distribution of samples it produces, when projected on to the flat MDP, is as close as possible to the "target distribution," which is the projection of the uniform random sampling procedure described above running on the hierarchical MDP. To derive the target distribution, sup-

pose that while uniformly sampling in the hierarchical MDP, every time we see $((c, s), a)$ where a is a primitive action, we record it. Let $P(s, a)$ be the asymptotic distribution over (s, a) we obtain in this manner. That is, the target distribution is $P(s, a)$. We have that $P(s, a) = P(a|s)P(s)$, where $P(s) = \sum_c P^*((c, s))$. Here $P^*((c, s))$ denotes the stationary state occupancy distribution of the uniform random policy on the hierarchical MDP. Further, $P(a|s) = \sum_c P(a, c|s) = \sum_c P(a|c, s)P(c|s)$. This leads to Eq. 1, where m is an index over the set of subtasks in the stack c, $k(m, s) = |\{x \mid \textbf{parent}(x) = \textbf{parent}(m) \wedge s \notin T_x\}|$ where T_x is the set of terminal states for subtask x.

$$P(c|s) = \prod_{m \in c} \frac{1}{k(m, s)} \tag{1}$$

The probability of primitive actions given a hierarchical state is shown in Eq. 2 where $l(a, (c, s)) = |\{x \mid a \in A_x \wedge x = \textbf{top}(c) \wedge s \notin T_x\}|$, where **top** equals the subtask at the top of the callstack c.

$$P(a|(c, s)) = \frac{1}{l(a, (c, s))} \tag{2}$$

Combining both equations gives Eq. 3.

$$P(a|s) = \sum_c P(a|(c, s))P(c|s) = \sum_c \frac{1}{l(a, (c, s))} \prod_{m \in c} \frac{1}{k(m, s)} \tag{3}$$

As a sanity check, consider a flat MDP. This flat MDP can be transformed into a hierarchy with a single root task and every primitive action as a child of the root task. This hierarchy tells the agent that all tasks are relevant to the root task and that all are equally likely to be relevant. Essentially it contains no information that the flat MDP does not have. Therefore, we would expect the projected hierarchical distribution to be uniform over the state-action space. Using the Eq. 3 and observing that $P^*((c, s)) = P^*(s)$, we see this is indeed the case.

We now observe that for every state in the MDP, the target action distribution has these properties:

1. Actions only available via terminated subtasks have zero probability of being sampled.
2. The probability of an action being chosen is weighted by how many paths in the hierarchy there are to that action.
3. The probability of an action being chosen is weighted by its depth in the hierarchy.

The first property is desirable because in general, subtasks are designed to be terminated when, for a given state, they are irrelevant. The second property is also desirable, because if an action is available in many subtasks, then that action can help complete multiple subtasks. The third property is beneficial as the root subtask represents the original MDP. Therefore, actions closer to the root are more directly relevant to the original task versus those deep down in the hierarchy.

It is important to note that this hierarchically projected distribution can be arbitrarily different from the flat uniform random sampling distribution. In states where there are actions unreachable in the hierarchy, the unreachable actions will have zero probability of being sampled, whereas in the flat uniform random distribution they will have a non-zero probability. The weighting due to actions appearing multiple times and an action's depth in the hierarchy can make them different even in states where there are no unreachable actions.

We now present three sampling strategies that could be used and analyze their properties with respect to the target distribution. These sampling strategies correspond to the *Sample-Hierarchy* function on line 3 of Algorithm 2.

3.2 Random hierarchical policy (RHP) sampling

An intuitively plausible sample collection strategy would be to simply run a hierarchical policy consisting of a uniform random policy for each subtask. This would be the closest analog to running a uniform random policy on the flat MDP. Thus each subtask takes uniform random actions until termination, and we keep track of the primitive state-action pairs seen during the process.

Though intuitive, we can prove that the resulting state/action distribution can actually be arbitrarily different from the target distribution. Let $N_{M_i}(s)$ be the expected completion time of the subtask M_i (steps before reaching a termination condition) when starting from state s and running a uniform random policy.

Theorem 1 *If the completion times, $N_{M_i}(s)$, of the subtasks are non-uniform, then the distribution of the set of state-action pairs sampled by RHP sampling may have arbitrary KL-divergence with respect to the target hierarchical projection distribution in the limit of arbitrary samples.*

Proof (*Sketch*) The intuition is that the probabilities of a child task being chosen are biased by the completion times. Therefore, the probability of state-action pairs appearing in the sample set will differ from the target $P(s, a)$. The completion times can be arbitrarily long, so that the difference between the actual distribution and the target distribution can be made arbitrarily different. Please see the appendix for details. □

We verify this result in our experiments. Empirically, this naïve strategy does not work well at all (i.e. LSPI produces very poor policies with these samples), in line with the theory. Thus we need to consider other sampling strategies.

3.3 Weakly polled sampling

The problem with using a random hierarchical policy is essentially that it can take too long to exit a subtask, thus biasing the samples collected. One way to fix this might be to add an additional "exit subtask" action to every subtask. This "exit subtask" action absorbs the original exits, and can be called at any time by the agent. When called, the agent immediately returns to the parent subtask. We call this approach "weakly polled sampling." Algorithm 3 shows the pseudo-code for this procedure. Line 11 modifies the action selection to select an action from the union of the set of children actions and the early exit action. Lines 12–14 ensure that if the early exit action was chosen then the function will exit early.

This strategy is better behaved than RHP sampling in that the sampling distribution produced does not diverge arbitrarily from the target. Nonetheless, it does not match it either. This is primarily because the extra action introduced slightly modifies $P(c|s)$ and $P(a|c, s)$, which leads to a slightly different distribution than the target distribution.

Theorem 2 *The distribution of the set of state-action pairs sampled by weakly polled sampling will have a nonzero KL-divergence with respect to the target hierarchical projection distribution in the limit of arbitrary samples.*

Proof (*Sketch*) The key idea is that each callstack probability is now $P(c|s) = \prod_{m \in c} \frac{1}{k(m,s)+1}$ and the probability of an action being chosen is now $P(a|(c, s)) = \frac{1}{l(a,(c,s))+1}$. Therefore, the sample distribution is different from the target distribution, but by a small amount. Please see the appendix for details. □

Algorithm 3 Weakly Polled Sampling

```
1: procedure WeaklyPolledSampling(M)        ▹ Implementation of SampleHierarchy from Algorithm 2.
2:     ▹ M : hierarchy for the MDP
3:     D ← ∅                                 ▹ D is the set of samples collected from this subtask
4:     M_i ← root(M)
5:     while subtasks in callstack not finished and more samples desired do
6:         if M_i is a primitive action then
7:             D ← collect primitive sample (s, M_i, r, s')
8:             M_i ← parent(M_i)
9:         else
10:            children ← all children of M_i where s is not a terminal state
11:            M_j ← random(children ∪ exitAction)     ▹ Select a random child action
12:            if M_j is exit action then
13:                M_i ← parent(M_j)
14:            end if
15:        end if
16:    end while
17:    return D
18: end procedure
```

3.4 Polled sampling

Instead of altering the action distribution by introducing optional exit actions, an alternative is to force an exit at each state in each subtask. Thus, to collect each sample, we restart at the root of the task hierarchy. We uniformly sample a task stack from the set of stacks where the current state is non-terminal in all subtasks of the stack. Then we sample an action available at that state from the subtask at the top of the stack. We call this sampling strategy "polled sampling" because it is analogous to the "polled execution" model of a hierarchical policy in MAXQ (Dietterich 2000). (This is also why we call the previous sampling method "weakly polled", as at each action selection step the agent has a chance of returning to just its parent instead of the root and re-selecting a path. But unlike polled sampling, this behavior is probabilistic.) In normal execution, the agent runs the selected subtask to completion. In polled execution, each time an action is selected, the agent starts at the root of the hierarchy and traverses the tree downwards until a leaf node is reached. Algorithm 4 shows the polled sampling algorithm pseudo-code.

It is useful to note that both weakly polled and polled sampling assumes that subtasks in a task hierarchy are *interruptible*, i.e. a subtask can be exited at any time and a new subtask can begin execution. MAXQ hierarchies possess this property, as shown by the polled execution model of MAXQ. Recent work has also considered interruptible options hierarchies (Comanici and Precup 2010).

Unlike the previous two approaches, we show that polled sampling will asymptotically converge to the target distribution.

Theorem 3 *The distribution of the set of state-action pairs sampled by Polled sampling will have zero KL-divergence with respect to the target hierarchical projection distribution in the limit of arbitrary samples.*

Proof At each step, the agent starts at the root and selects a path in the hierarchy through non-terminated subtasks to a primitive action. The segments in the selected path are selected independently at each step with uniform probability. This means that $P(c|s)$ and $P(a|(c, s))$ are the same as the target distribution. Likewise, the total probability of seeing a state is the probability of uniformly sampling any stack consistent with its predecessors times the

Algorithm 4 Polled Sampling

```
1: procedure POLLEDSAMPLING(M)
2:     ▷ M : hierarchy for the MDP
3:     D ← ∅
4:     while collecting samples do
5:         M_i ← root(M)
6:         while M_i is not primitive action do
7:             children ← all children of M_i where s is not a terminal state
8:             M_i ← random(children)                ▷ Select a random child action
9:         end while
10:        D ← D ∪ (s, M_i, r, s')
11:    end while
12:    return D
13: end procedure
```

action selection probability times the transition function. Each of these quantities is the same as in the target distribution. This product therefore converges to the occupancy distribution $P^*((c, s))$. Therefore, in the limit polled sampling will have zero KL-divergence with respect to the target distribution. □

Out of the three sampling strategies above, only the Polled sampling algorithm is guaranteed to converge with zero KL-divergence from the target distribution. Weakly polled sampling is expected to converge with a small KL-divergence. Random hierarchical policy sampling is expected to be arbitrarily different from the target distribution in any task hierarchy where one subtask takes a lot longer than another to complete when choosing random actions. Unfortunately, many task hierarchies have at least one task that takes a long time to complete using only random actions. Therefore, in practice one would expect polled sampling to perform best, weakly polled sampling to perform slightly worse and random hierarchical policy sampling to perform poorly. As will be shown in the experiments, this is indeed the observed performance.

4 Derived samples

The sampling strategies presented above generate samples consistent with the task hierarchy; however, just providing a set of samples consistent with the hierarchy to an offline learner may not be sufficient to learn a good policy. The learner may also need additional information about which actions were *purposefully not sampled* due to the hierarchy constraints. In the next section we discuss how to generate this additional information in a form the learner can utilize.

Additionally, many hierarchical decompositions include additional information beyond which primitive actions are available in a state. In the next section we also present a method for augmenting the sample set using this additional information. We call these generated samples "derived samples" because they are derived from the set of samples collected by the strategies above.

4.1 Inhibited action samples

When using the hierarchical sampling approaches above, the distribution of samples will not contain state-action pairs that are inconsistent with the hierarchy. This presents a problem for the offline learning algorithm: how does the learner tell the difference

between a state-action pair that is purposefully not sampled, and a state-action pair that has been left out due to the sample set being finite? Without indicating which state-action pairs are absent from the sample set purposefully, the learner may be misled, and choose purposely excluded state-action pairs for the final policy! We do not want to have to sample these purposefully avoided actions as the hierarchy has restricted them for some reason (they could incur a penalty, be dangerous to the agent or its environment, etc.). In this section we detail a method for augmenting the sample set with special "inhibited action samples" which prevent the learner from choosing a purposefully not-sampled state-action pair without have to actually test the state-action pair in the environment.

For an example where this problem arises, consider the Taxi-World domain, where an agent representing a taxi must navigate a grid-world in order to pickup and drop-off passengers from designated locations (more details in Sect. 6). In any state where the agent is not on a designated pickup/drop-off location, the pickup and drop-off actions are inhibited by the hierarchy. This means that at such locations, samples of the pickup and drop-off actions will not appear in the sample set collected by any of the sampling methods. While this is desired because the agent does not waste time sampling these useless actions, the learner still needs to be told that these unsampled actions are bad choices. Otherwise, the learner will see the move actions as having a reward of -1 and the the pickup and drop-off actions as having a reward of zero. This can cause the policy chosen by the learner to map all states where the agent is not on a pickup/drop-off location to either the pickup or drop-off action, which is clearly the wrong policy.

To address this without requiring modification of the offline learning algorithm, we provide additional samples to the learner which tell it to never choose the actions inhibited by the hierarchy at a given state. We call these samples "inhibited action samples."

For states where the non-inhibited actions all have positive rewards, the inhibited action samples are not necessary, because a reward of zero for the unsampled inhibited actions will be less than the positive rewards of all of the other actions. However, in such cases, explicitly providing the inhibited action samples may still speed up convergence.

To generate inhibited action samples, for every state observed when collecting samples with the hierarchical sampling approaches, we consult the hierarchy to see if there are any inhibited actions at that state. These are actions that are unavailable at all subtasks where the state is non-terminal. For each such action (if any) we generate a new sample with the same state, but a special reward value, $R_{min} = \frac{r_{min}}{1-\gamma}$, where r_{min} is a value lower than the minimum reward for any action in our sample. Thus selecting such an action immediately yields a feedback signal that is worse than any possible sequence of other actions, so that the agent never selects this action for its final policy.

Algorithm 5, lines 3 through 13, shows the pseudo-code for the inhibited action sample generation.

4.2 Abstract samples

Certain task decompositions, such as MAXQ task hierarchies, provide additional state abstraction information. This state abstraction information is defined as a function for each subtask, which specifies the set of variables necessary to completely determine the value function of any policy in that subtask. The key criterion is the "Max Node Irrelevance" condition (Dietterich 2000). Assume that the state can be factored into the set of relevant state variables, X, and the irrelevant state variables, Y. That is for a given state, s, it can be decomposed as follows: $s = (x, y)$. Max Node Irrelevance guarantees that the transition function is factorable

Algorithm 5 Generate Derived Samples

```
1: procedure GENERATEDERIVEDSAMPLES(D, M, γ)
2:     ▷ D: Set of real samples (s, a, r, s'), M: hierarchy, γ: discount factor
3:     ▷ First generate the inhibited action samples
4:     R_min ← r_min / (1−γ)                    ▷ r_min is the minimum reward of all samples in D
5:     D_inhib ← Ø
6:     for sample d ∈ D do
7:         for all actions a do
8:             if a is unreachable from root in state s from sample d then
9:                 D_inhib ← D_inhib ∪ {(s, a, R_min, s')}    ▷ s and s' match the states in sample d
10:            end if
11:        end for
12:    end for
13:    D ← D ∪ D_inhib
14:    ▷ Now using the real and inhibited action samples, generate abstract samples.
15:    D_abs ← Ø
16:    for sample (s, a, s', r) ∈ D do
17:        Y ← all vars meeting Max Node irrelevance for action a and hierarchy M
18:        for each combination of values, y, generated from Y do
19:            s_new, s'_new ← s, s'
20:            for y_i ∈ y do
21:                s_new(y_i), s'_new(y_i) ← y_i      ▷ Change parts of the state deemed irrelevant
22:            end for
23:            D_abs ← D_abs ∪ {(s_new, a, r, s'_new)}
24:        end for
25:    end for
26:    return D ∪ D_abs
27: end procedure
```

as shown in Eq. 4 and that the reward function is independent of the irrelevant variables as shown in Eq. 5.

$$P(x', y'|x, y, a) = P(y'|y, a)P(x'|x, a) \tag{4}$$

$$R(x', y'|x, y, a) = R(x'|x, a). \tag{5}$$

Our sampling approaches can take advantage of this information. In particular, given the state abstraction functions at a subtask, we can generate extra “abstract samples” whenever we generate any “real” sample.

Abstract samples are a way of using the abstract state functions to generalize the collected samples across irrelevant state variables. If a state variable is irrelevant, then we know that changing the value of that variable has no effect on the resulting state and reward when an action is taken. Thus for each sample collected in some subtask, we find possible irrelevant variables by examining all subtasks where the state in this sample is non-terminal and the action taken is available, and taking the intersection of their irrelevant variable sets. We then generate new samples by changing the values of irrelevant variables, leaving the rest of the sample unchanged. Abstract samples do not add any information that the agent could not eventually find through normal sample collection, however, they significantly speed up the collection of this information. This procedure leads to the algorithm shown in Algorithm 5 lines 15 through 25. It is important to note that when, generating the abstract samples, we use both the real collected samples and the generated inhibited action samples.

5 OHRL and policy optimality

Because two of our sampling strategies will cover all (state, action) pairs consistent with the task hierarchy arbitrarily often in the limit, we have the following result.

Proposition 1 *Assume we have (i) an MDP M, (ii) an offline learner L capable of finding the optimal policy using enough samples collected via uniform random sampling from M, and (iii) a hierarchy T for M in which each (a) subtask is interruptible, and which provides (b) a list of primitive actions allowed by each subtask in each state and (c) state abstraction for each subtask. Then with enough samples collected via Polled or weakly polled sampling with derived samples, L will find a policy in M that has value greater than or equal to the value of the hierarchically optimal policy in T.*

Proof For every (c, s) in T and every a available to s in T, there is a non-zero probability of $((c, s), a)$ being sampled. For any $((c, s), a)$ not valid in the hierarchy there is zero probability of it being sampled. Therefore, with enough samples, every $((c, s), a)$ consistent with T will be sampled arbitrarily many times. This means that L will be able to accurately estimate the value of each state-action pair, which will allow it to find the optimal policy in M consistent with the constraints of T.

Inhibited action samples guarantee that if a is not a valid action in s for all c, it will never be chosen in s by the final policy returned by L. A hierarchical policy in T has at least the same constraint. Thus policies in the space searched by L have at least as many actions available in any state as hierarchical policies in T. Therefore, the optimal policy returnable by L must have value at least as good as the hierarchically optimal policy.

Abstract samples do not affect the policy in the limit as any abstract sample could be collected via normal sample collection. With an arbitrarily large number of samples every possible abstract sample will have already been collected via normal sampling. □

The assumptions in the result above are not difficult to satisfy. LSPI is an offline learner that can find the optimal policy in an MDP given enough samples and a good state representation. As we have discussed above, different task hierarchy frameworks exist that possess the interruptability property. For task hierarchies, it is usually straightforward to find all primitive actions executable in a state. In MAXQ, one simply needs to recursively traverse the task graph to find the primitive action nodes that are the children of any subtask. For something like an options hierarchy, the options are the subtasks and the only "allowed" primitive action of an option would be the one chosen at that state by that option (Stolle and Precup 2002). State abstraction functions are generally assumed to be specified along with the hierarchy.

We note again that the final policy found by OHRL will be a policy in the original flat MDP. Global optimality cannot be guaranteed in every case because it is always possible to provide the agent with a "bad" hierarchy which would prevent it from discovering useful state-action pairs.[1] The benefit is that this flat MDP policy is guaranteed to have a value at least equal to the value of a hierarchically optimal policy for the given hierarchy. MAXQ can only guarantee recursively optimal policies when pseudo-rewards are not used, and in general the value of a recursively optimal policy can be arbitrarily worse than the value of a hierarchically optimal policy. In some cases, as we show in our experiments, the hierarchical sampling algorithm can find a policy with value *greater* than the value of a hierarchically optimal policy for the given hierarchy, even with samples generated from the hierarchy.

[1] Of course this same issue exists in all other hierarchical RL methods.

6 Empirical evaluation

In this section, we evaluate several hypotheses empirically using a combination of hierarchical sampling and LSPI, which we refer to as HS-LSPI. The hypotheses are:

1. HS-LSPI can improve the rate of convergence to the optimal policy when compared to flat learning methods, like LSPI and Q-learning.
2. HS-LSPI can improve the rate of convergence to the optimal policy when compared to hierarchical learning methods, like MAXQ.
3. HS-LSPI can converge to the hierarchically optimal policy, even when hierarchical learning methods, like MAXQ, require pseudo-rewards.
4. HS-LSPI can sometimes yield policies that are *better* than hierarchically optimal, even using samples consistent with a provided task hierarchy.
5. Derived samples improve the rate of convergence of HS-LSPI with Polled sampling to the optimal policy. However, derived samples alone cannot fix a poor sampling policy, such as RHP sampling.
6. The KL-divergence of the sample distribution generated by the HS-LSPI variants (Random Hierarchical Policy, weakly polled and polled sampling) to the target distribution will match the theoretical predictions.
7. HS-LSPI is competitive with baseline approaches in terms of computational efficiency.

We use two problem domains in our experiments: Taxi-World (Dietterich 2000) and Resource-Collection (Cao and Ray 2012).[2] We use several variations of Resource-Collection depending on the hypothesis being evaluated. In Taxi-World (500 states, 6 primitive actions), the agent controls a taxi that must pick up and deliver passengers from different sources to different destinations on a map. Each move has a 20% failing and moving the agent in a perpendicular direction. In Resource-Collection ($\sim$12,288 states, 7 primitive actions), the agent must navigate to different trees and mines in a map in order to collect target amounts of gold and wood (this simulates the resource collection task in common strategy games such as Warcraft). Collected gold and wood must be deposited at townhalls. Further, mines and trees are finite resources and disappear after they are exhausted, and each move action has a 20% failing and moving the agent in a perpendicular direction.

All experiments used $\gamma = 0.9$ and ϵ-greedy exploration policies with exponential decay. MAXQ used an $\epsilon_{start} = 0.5$, $\epsilon_{decay} = 0.99$, $\epsilon_{min} = 0.1$ with an exponentially decayed learning rate $\alpha_{start} = 0.25$, $\alpha_{decay} = 0.99$, $\alpha_{min} = 0.1$. Flat Q-learning used $\epsilon_{start} = 0.5$, $\epsilon_{decay} = 0.95$, $\epsilon_{min} = 0.1$ and $\alpha_{start} = 0.5$, $\alpha_{decay} = 0.99$, $\alpha_{min} = 0.1$. The LSPI solver used a tolerance of 0.001.

The baseline approaches we evaluate are MAXQ, standard Q-learning and standard LSPI. We show results for two variants of HS-LSPI: weakly polled and polled sampling with derived samples. We do not show results for random hierarchical policy sampling for most experiments because it generally does poorly (as our theory indicates). We plot learning curves where after a certain number of sample collection/learning steps, the policies of each agent were recomputed/frozen and tested for several episodes. We measure the average cumulative reward obtained by the current policy and plot the reward achieved by the best policy found thus far.[3] We repeat the entire process five times and average the resulting curves. High learning rates were purposefully picked for MAXQ and Flat Q-learning in order to make

[2] The task hierarchies for all domains are shown in the appendix.

[3] The results corresponding to the cumulative reward of the updated policy at each step (rather than the best policy) are in the appendix.

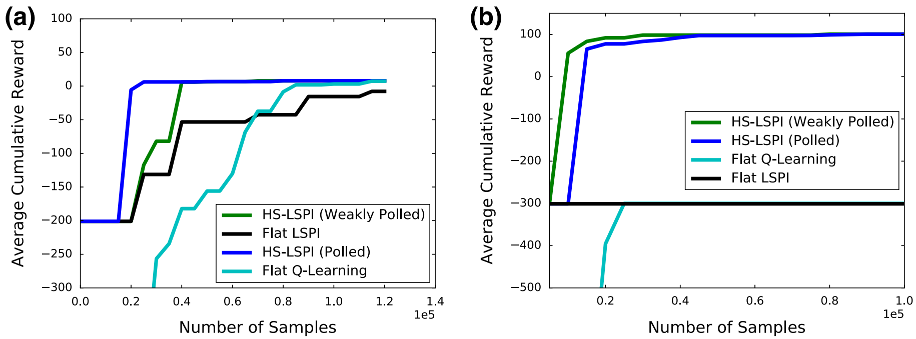

Fig. 1 Standard LSPI and Q-Learning versus HS-LSPI methods on Taxi-World (**a**) and Resource-Collection (**b**). Each data point is averaged over five runs.

the convergence rates as competitive as possible with the hierarchical sampling algorithms. Therefore, the raw learning curves shown in the appendix, will appear to have high variance. All methods use a tabular representation with no state-approximation or extra features. While all of the methods are capable of using state approximation, tabular representations were used to eliminate the possibility that poor performance was due to state approximation and not the inherent properties of the method.

6.1 Hierarchical sampling versus flat learning

The first hypothesis we evaluate is that HS-LSPI can improve the rate of convergence to the optimal policy when compared to flat learning methods.

Figure 1 compares the performance of HS-LSPI to standard LSPI and Q-learning on Taxi-world and Resource-Collection. In both domains, HS-LSPI performs better than the alternatives. The difference is more significant for Resource-Collection, not surprisingly since it is a much larger problem. Furthermore, in this domain, there are many more opportunities to generate abstract samples and inhibited action samples. While standard LSPI must explicitly figure out that the trees have no effect on the collect gold actions, the hierarchical sampler can generate this information using abstract samples. In addition, the hierarchical sampler will not waste samples on actions deemed irrelevant by the hierarchy, such as deposit actions when the agent is not holding any resource.

We also see from these results that Polled sampling is generally superior to or competitive with weakly polled sampling for HS-LSPI, as is also indicated by our theoretical results. Hence, for the remaining experiments, we only show results for Polled sampling.

6.2 Hierarchical sampling versus hierarchical learning

We next compare HS-LSPI with Polled sampling to hierarchical learning with MAXQ-Q learning. This is an online hierarchical learning algorithm. Figure 2 shows the results for Taxi-World and Resource-Collection

We first note that in both domains, the recursively optimal policy is also hierarchically and globally optimal. Thus MAXQ can achieve optimality given enough samples, so we expect no asymptotic difference between these approaches. However, the rate of convergence to the optimal policy differs for the different approaches. In particular, we see that HS-LSPI with Polled sampling converges faster to the optimal policy than MAXQ. Again, the difference

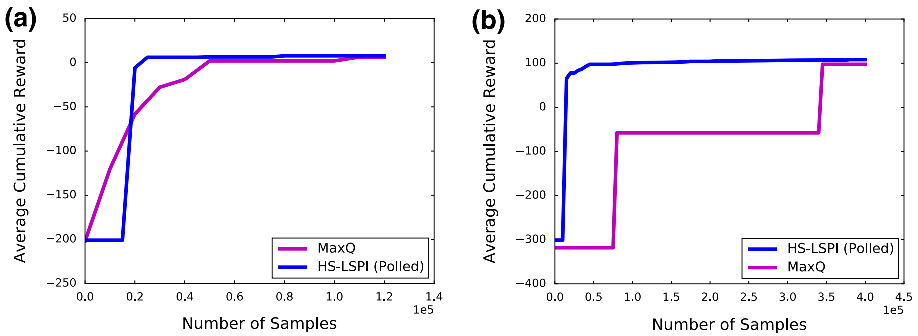

Fig. 2 MAXQ-Q learning versus HS-LSPI with Polled sampling on Taxi-World (**a**) and Resource-Collection (**b**). Each data point is averaged over 5 runs.

is particularly significant in the larger Resource-Collection domain. The reason for this difference is likely that MAXQ-Q performs local policy updates focused on the current subtask for each new sample, whereas for any given sample size, HS-LSPI with Polled sampling collects samples from all subtasks and performs a global optimization, making progress in the entire task.

6.3 Hierarchical sampling versus hierarchical learning with pseudo-rewards

We next compare HS-LSPI with Polled sampling to hierarchical learning with MAXQ on domains where the recursively optimal policy is *not* the hierarchically optimal policy. To do this, we use Pseudorewards-RC, a variant of the Resource-Collection domain. Pseudorewards-RC has two townhalls—one with an extra negative reward for depositing gold, and one with an extra negative reward for depositing wood. To get the best rewards, the agent must learn which resource each townhall prefers. In the task hierarchy, this corresponds to the Deposit task having multiple exits (one for each townhall). However, picking the correct exit consistently requires a pseudo-reward to be provided for MAXQ. Figure 3a shows these results. For comparison, we also show results for flat Q-learning.

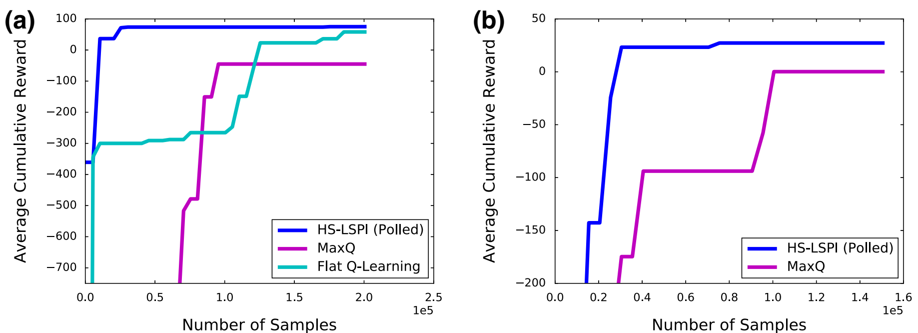

Fig. 3 **a** Standard Q-learning and MAXQ-Q learning versus HS-LSPI with polled sampling on Pseudorewards-RC . All data points are averaged over five runs. **b** MAXQ-Q learning versus HS-LSPI with Polled sampling on Parallel-RC . All data points are averaged over 5 runs.

From these results, we see that not only does HS-LSPI converge faster to the optimal policy, it reaches an asymptotically better solution than MAXQ. The found solution is identical to the asymptotic solution found by flat Q-learning, confirming that this is the optimal policy. It would be possible to enable MAXQ to reach hierarchical optimality in this domain if the pseudo-rewards were correctly specified. However, as we have argued earlier, this is difficult to do in practice without detailed knowledge of the optimal policy. As well, mis-specified pseudo-rewards can hurt or even prevent convergence. In contrast, HS-LSPI completely sidesteps pseudo-rewards. This approach still learns the hierarchically optimal policy by using the hierarchy to *sample*, and using a global optimization strategy to learn. Our results indicate this is a significantly better way to learn policies using task hierarchies.

6.4 Hierarchical sampling can learn policies better than hierarchically optimal

We next show that for some domains, it is possible for HS-LSPI to learn policies better than hierarchically optimal when using samples generated from a given task hierarchy. To show this, we use Parallel-RC, a variant of the Resource-Collection domain. In Parallel-RC, the agent can simultaneously carry gold and wood, allowing a good globally-optimal policy to exploit concurrency in gold collection and wood collection. The task hierarchy remains the same as in Resource-Collection, and the recursively optimal policy is still hierarchically optimal (pseudo-rewards are not needed). These results are shown in Fig. 3b.

From these results, we observe that HS-LSPI converges faster than MAXQ, and more importantly, to a better final policy. Since in this domain the recursively optimal policy is hierarchically optimal, the policy found by HS-LSPI is *better* than hierarchically optimal, though its samples are generated from the *same* task hierarchy that MAXQ is using to learn. The first reason for this seemingly paradoxical result is that MAXQ's standard learning algorithm does not allow subtasks to be interleaved. In this domain, however, because the agent can simultaneously carry gold and wood, the gold and wood collection subtasks can be concurrently active and yield a better policy. HS-LSPI does not possess this restriction. The second reason is that during sample collection with polled sampling, we *do* allow subtasks to be pre-emptively terminated. Thus we can collect samples where the net effect is that the agent simultaneously carries gold and wood, allowing LSPI to learn that this is a good strategy. Thus, in such scenarios, our approach will converge to policies that are better than hierarchically optimal (globally optimal in this case). Again, this result indicates that using the task hierarchy for sampling rather than learning yields the best sample complexity and optimality trade-offs.

6.5 Derived samples improve convergence rate

A key part of HS-LSPI is the addition of derived samples, including samples for inhibited actions and abstract samples, to the initial set of samples collected by polled sampling. In this experiment, we evaluate the extent to which this step helps the rate at which HS-LSPI converges. In this experiment, we show results for Taxi-World (the results for Resource-Collection are similar). These are shown in Fig. 4.

Figure 4a shows that the additional abstract samples have a large effect on the convergence rate of the policy. Essentially, the extra abstract samples add information about states that the agent has not visited, based on abstraction information provided by the task hierarchy, and thus can significantly improve the policy even with few "real" samples.

The inhibited action samples have an even larger effect. Without these samples, as expected, the policy actually does not converge. This is due to the effect of having nega-

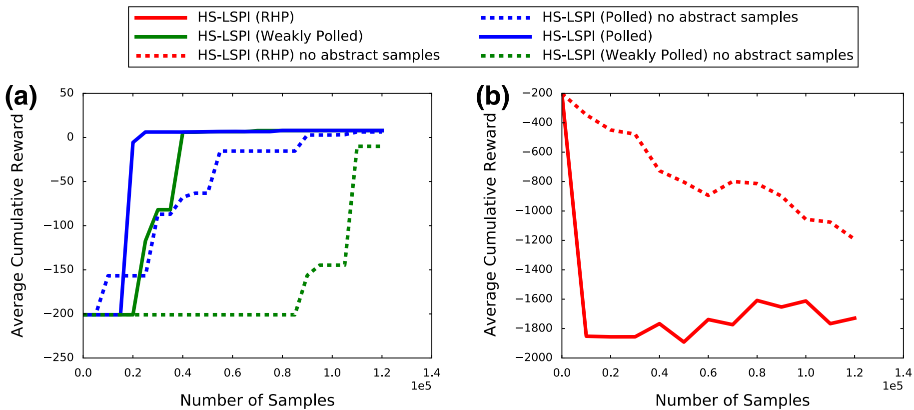

Fig. 4 **a** HS-LSPI polled sampling with inhibited action samples versus HS-LSPI polled sampling with inhibited action *and* abstract samples on Taxi-World . All data points averaged over 5 runs. **b** HS-LSPI RHP sampling with abstract samples versus HS-LSPI RHP sampling without abstract samples.

tive action rewards combined with unsampled actions as discussed in Sect. 4.1. Essentially, the learning algorithm associates the actions unsampled, because the hierarchy inhibits them, as having rewards of zero. The other actions allowed by the hierarchy have negative rewards, so the learner is misled into thinking the unsampled actions are the best choice, when they are not. Adding the inhibited action samples indicates to the learner that these unsampled actions actually have a very bad reward, and so the learned policy does not include them.

Figure 4b shows that the additional abstract samples are not sufficient by themselves to increase convergence rate. In fact, with a poor sampling strategy, they can make the policy worse! When using the RHP sampling method, even with abstract samples, the agent is very unlikely to sample pickup actions in the Taxi-World domain. This leads to the convergence of the Q-values of the move actions to a small negative value, the convergence of the Q-values for the drop-off action to a large negative value and pickup action Q-values to remain at zero. Hence, the resulting "best" policy is to just call pickup everywhere, leading to poor performance. Thus in general, the derived samples cannot "fix" a poor sampling strategy by themselves.

6.6 Verification of theoretical KL-divergence results

In the previous section, we presented theoretical results about the asymptotic convergence properties of each of the sampling variations in HS-LSPI. We showed that in the limit, polled sampling will converge without bias to the target hierarchical projection distribution. We also gave KL-Divergence results for weakly polled sampling and random hierarchical policy sampling. These results are illustrated in Fig. 5, using Taxi-World For these results, the exact value of the target hierarchical projection distribution was calculated by enumerating every state possible and using the hierarchy with Eq. 3.

From the figure, we see that as predicted by the theory, random hierarchical policy sampling has a large KL-divergence with respect to the target. Note that while there is no upper limit on the RHP KL-divergence for an arbitrary environment, it will be finite for a particular environment. Intuitively, this is because the navigate tasks take a lot longer to complete than the pickup and drop-off tasks. There is some divergence in the weakly polled distribution, however, this bias is expected to be small because the probabilities of each action being

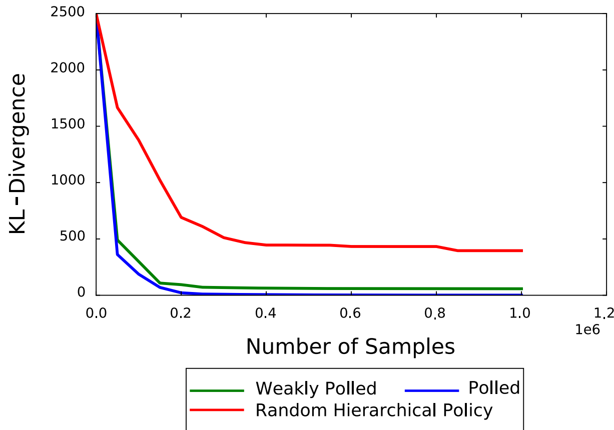

Fig. 5 KL-Divergence curves for polled sampling, weakly polled sampling, and random hierarchical policy sampling on Taxi-World .

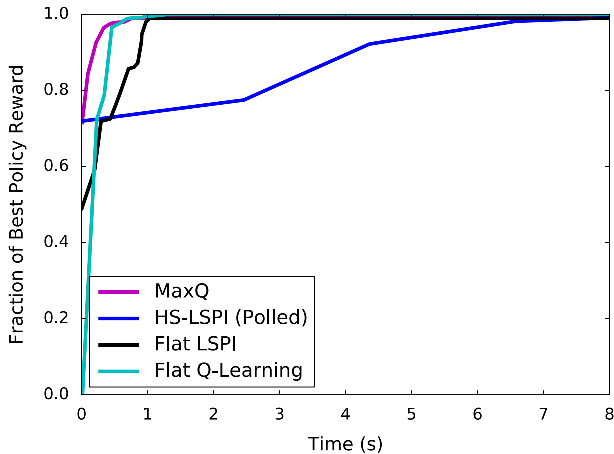

Fig. 6 Fraction of best policy reward achieved versus total training time for the different algorithms on Taxi-World . Each data point averaged over 5 runs.

chosen are only slightly decreased by adding in the early exit action. Finally, as expected, polled sampling asymptotically converges to the target distribution.

6.7 HS-LSPI is computationally competitive with baselines

Figure 6 compares the computational costs for various algorithms on the Taxi-World domain. The graph was constructed by recording total training time (including sample collection and in the case of HS-LSPI (Polled), derived sample generation). The y-axis shows the fraction of the best policy's reward achieved by each agent versus the amount of training time in seconds. The closer the curve is to the left of the graph, the better the algorithm performs in the trade-off between policy quality and training time. From the graph we observe that MAXQ takes the smallest amount of time to converge, followed by flat Q-learning and flat LSPI. In general, as expected, the offline algorithms take more computation time than the online

algorithms. HS-LSPI takes longer than the other methods to run. However, the difference is not large.

The main reason for the increased runtime of HS-LSPI is because the additional derived (especially abstract) samples greatly increase the number of samples processed in each iteration of the LSPI solver. At the moment, we generate all possible abstract samples for any real sample. Thus a direction for future work is to investigate how to improve the computational cost of the algorithm by limiting the number of abstract samples generated, while still maintaining the sample efficiency in converging to the best policy. It is useful to note that, especially for offline learners such as ours, sample efficiency is typically a much more desirable algorithm characteristic than computational efficiency, although of course the latter cannot be neglected.

To summarize, given the results described above, we argue that HS-LSPI effectively leverages synergies that exist between offline learning and task hierarchies. Using a task hierarchy to sample and then using an offline learning approach couples the reduced sample complexity offered by the hierarchy's constraints with the ability to find policies that are as good as hierarchically optimal policies or better. This combination of properties therefore offers an effective alternative to online learning in HRL, and an effective way to augment sample collection for offline RL algorithms such as LSPI.

7 Conclusion

In this work, we build upon the observation that there are synergies we can exploit between offline RL and hierarchical RL. In particular, we use a task hierarchy to guide the sample collection process for offline RL. We derive suitable sampling strategies and study the asymptotic properties of the sample distributions they generate. We then augment these samples using additional information from the task hierarchy and use this set as input to an offline learner, LSPI. Our empirical results show that our approach outperforms LSPI using standard sampling, as well as online HRL approaches, in terms of rate of convergence to optimal policy and ability to find better policies. These results indicate that (1) using a task hierarchy can speed up the convergence of an offline RL algorithm and (2) we can get most of the benefit of a task hierarchy through sampling from it while avoiding suboptimalities in the policy space.

Appendix A: full proofs

Random hierarchical policy sampling

Theorem 1 *If the completion times, $N_{M_i}(s)$, of the subtasks are non-uniform, then the distribution of the set of state-action pairs sampled by RHP sampling may have arbitrary KL-divergence with respect to the target hierarchical projection distribution in the limit of arbitrary samples.*

Proof Let $N_{M_i}(s)$ be the expected completion time of the subtask M_i (steps before reaching a termination condition) when starting from state s and running a uniform random policy. Using the completion times for each subtask, the probability of a subtask being on the current callstack can be calculated. Each time the agent chooses a subtask, on average, it will sample it $N_{M_i}(s)$ times. This biases the distribution of subtasks on the callstack to those where the

child subtask has a longer completion time. Equation 6 shows the probability of a subtask being on the callstack under these assumptions.

Let M_j be a subtask in the callstack. Let M_i be the subtask one up in the callstack from M_j. Let $child(M_i, s)$ be the children of subtask M_i for the state s, which includes M_j. Then the probability of a subtask M_j being on the callstack in state s under RHP sampling, $P_{RHP}(M_j, s)$, is as follows:

$$\begin{aligned} P_{RHP}(M_j, s) &= \frac{\frac{N_{M_j}(s)}{|child(M_i,s)|}}{\sum_{M_k \in child(M_i,s)} \frac{N_{M_k}(s)}{|child(M_i,s)|}} \\ &= \frac{N_{M_j}(s)}{\sum_{M_k \in child(M_i,s)} N_{M_k}(s)}. \end{aligned} \tag{6}$$

The individual probabilities of a subtask being on the callstack can then be combined to find the probability of a particular callstack being sampled via RHP sampling. In RHP the subtasks are chosen independently from their siblings, so the total probability of a callstack c in a state s via an RHP sampling policy, $P_{RHP}(c, s)$, is just the product of the probability of each subtask being in the callstack. Equation 7 shows this probability.

$$\begin{aligned} P_{RHP}(c, s) &= P((M_0, M_1, \ldots, M_{|c|}), s) \\ &= P(M_0, s)P(M_1, s) \ldots P(M_{|c|}, s) \\ &= P(M_0, s) \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k \in child(M_{i-1},s)} N_{M_k}(s)} \\ &= \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k \in child(M_{i-1},s)} N_{M_k}(s)}. \end{aligned} \tag{7}$$

Here $P(M_0, s) = 1$, because the root subtask is always in the callstack. $|c|$ is the number of subtasks in the callstack c.

Using $P_{RHP}(c, s)$, the probability of an action a being sampled, given state s, under RHP sampling, $P_{RHP}(a|s)$, can be written as shown in Eq. 8.

$$P_{RHP}(a|s) = \sum_{c \in C | a \in c} \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k \in child(M_{i-1},s)} N_{M_k}(s)} \tag{8}$$

where c is a callstack and C is the set of all callstacks possible in the MDP.

Rewriting this as the probability over the joint state-action space yields the form shown in Eq. 9.

$$P_{RHP}(a, s) = \sum_{c \in C | a \in c} \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k \in child(M_{i-1},s)} N_{M_k}(s)} \sum_{c} P^*_{RHP}((c, s)) \tag{9}$$

where $P^*_{RHP}((c, s))$ is the stationary distribution over the hierarchical state (c, s) via RHP sampling, which is a combination of callstack and state.

Using the KL-Divergence the difference between the probabilities of state-action pairs in the target distribution, $P(a, s)$, and the RHP distribution, $P_{RHP}(a, s)$ can be calculated. The result is shown in Eq. 10.

$$
\begin{aligned}
D_{KL}(P||P_{RHP}) &= \sum_{s\in S}\sum_{a\in A} P(a, s) \ln \frac{P(a, s)}{P_{RHP}(a, s)} \\
&= \sum_{s\in S}\sum_{a\in A} P(a, s) \\
&\quad \ln \left(\frac{P(a, s)}{\sum_{c\in C|a\in c} \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k\in child(M_{i-1}, s)} N_{M_k}(s)} \sum_c P^*_{RHP}((c, s))} \right).
\end{aligned} \tag{10}
$$

To prove that this is unbounded, substitute in the target distribution subtask probabilities. Let $k(m, s) = \{x \mid \mathbf{parent}(x) = \mathbf{parent}(m) \wedge s \notin T_x\}$ where T_x is the set of terminal states for subtask x. This leads to the KL-Divergence shown in Eq. 11.

$$
\begin{aligned}
&D_{KL}(P||P_{RHP}) \\
&= \sum_{s\in S}\sum_{a\in A} \left(\sum_{c\in C|a\in c} \prod_{i=1}^{|c|} \frac{1}{|k(M_i, s)|} \sum_c P^*((c, s)) \right) \\
&\quad \ln \left(\frac{\sum_{c\in C|a\in c} \prod_{i=1}^{|c|} \frac{1}{|k(M_i, s)|} \sum_c P^*((c, s))}{\sum_{c\in C|a\in c} \prod_{i=1}^{|c|} \frac{N_{M_i}(s)}{\sum_{M_k\in child(M_{i-1}, s)} N_{M_k}(s)} \sum_c P^*_{RHP}((c, s))} \right).
\end{aligned} \tag{11}
$$

The target distribution subtask probabilities depend on the number of children subtask + number of children primitive actions in each subtask. The RHP subtask probabilities depend on the completion times of each subtask. The completion time of a subtask is unrelated to its number of children. Let us assume for now that the following equation holds: $\sum_c P^*_{RHP}((c, s)) = \sum_c P^*((c, s))$. The only way the RHP and target distributions would be equal, is if every subtask had equal completion times for all its children. However, this is very unlikely to be the case and in general the completion times for a subtask can be arbitrarily long.

For example consider an MDP where one subtask involves navigating a 2D grid to a specific point. By increasing the distance of the agent's position to the termination point of the subtask, we can construct an MDP which has a navigate task with an arbitrarily high completion time. If the other tasks in the hierarchy remain constant in their completion times, we can invent an MDP with a task hierarchy, in which the navigate subtask takes arbitrarily longer to complete (on average) than the other subtasks in the hierarchy.

Because it is possible for every subtask to have one child with a completion time that is arbitrarily bigger than the other completion times, the target distribution and the RHP distribution can be arbitrarily different. Therefore RHP's KL-Divergence from the target distribution can be arbitrarily large. □

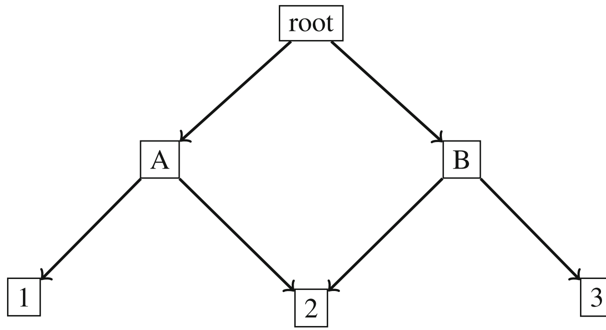

Fig. 7 Weakly polled example hierarchy

Weakly polled sampling

Theorem 2 *The distribution of the set of state-action pairs sampled by Weakly Polled sampling will have a nonzero KL-divergence with respect to the target hierarchical projection distribution in the limit of arbitrary samples.*

Proof The proof for weakly polled sampling depends on the hierarchy being modeled as a Markov chain. To illustrate how the Markov chain is derived from the hierarchy, consider the hierarchy in Fig. 7. Figure 8 shows the Markov chain version of this hierarchy. Each of the subtask nodes in the hierarchy are now nodes in the Markov chain. Shared nodes must be split so that a copy exists for each possible parent. This is because when transitioning back to the parent, the agent will always transition back to the parent which it originally came from. When the probabilities of sampling from one of the Markov chain nodes is calculated all the split nodes can be summed to get the probability of the original unsplit node.

In the original hierarchy the probability of choosing an edge was uniform over all of the children nodes. For this example, each edge has a probability of 0.5. In the Markov chain version, the probabilities of transitioning along a specific edge is uniform among all of the outgoing edges from the node. The number of outgoing edges is the number of children the node in the hierarchy has plus one for the new early exit edge. This shown in Eq. 12.

$$P(e_i) = \frac{1}{|outgoing(M_i)|} = \frac{1}{|child(M_i)| + 1}. \quad (12)$$

In this equation e_i is an edge in the Markov chain attached to subtask M_i. $P(e_i)$ is the probability of transitioning along an edge e while in the node M_i. $outgoing(M_i)$ is the set of all outgoing edges of node M_i, and $child(M_i)$ is the set of children of node M_i in the original hierarchy.

When analyzing a Markov chain, it is important to check if the chain is bipartite. A bipartite chain can be divided into two classes of nodes: even nodes and odd nodes. In a bipartite chain there is no stationary distribution over the nodes because of periodicity between the even and odd nodes (Lovász 1996; Aldous and Fill 2014). If at time t, the agent is in a node in the even class, then the probability of it being in an even node at time $t + 1$ is zero and the probability

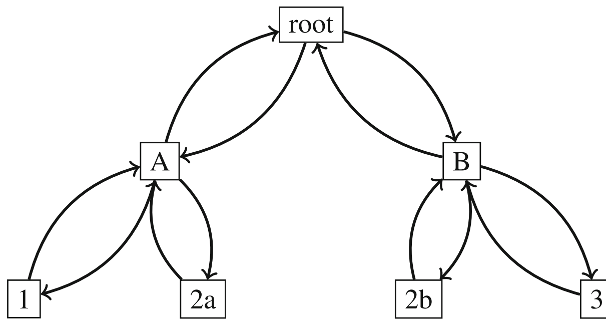

Fig. 8 Weakly polled example Markov chain

of it being in an odd node at time $t + 1$ is 1. More generally, the probability distributions become those shown in Eqs. 13 and 14.

$$\forall k \geq 0,\ P(even) = \begin{cases} 1 & \text{if } t + 2k \\ 0 & \text{if } t + 2k + 1 \end{cases} \tag{13}$$

$$\forall k \geq 0,\ P(odd) = \begin{cases} 0 & \text{if } t + 2k \\ 1 & \text{if } t + 2k + 1 \end{cases} \tag{14}$$

Even though a task hierarchy can be either bipartite or non-bipartite, for the sake of analysis, first consider non-bipartite hierarchies.[4] If a hierarchy is bipartite, then the nodes can be split into their even and odd classes and then analyzed using the same non-bipartite Markov chain stationary distribution equations.

For a non-bipartite Markov chain, the stationary distribution over the states is defined by the simple formula (Lovász 1996; Aldous and Fill 2014) shown in Eq. 15.

$$\omega(v) = \frac{C(v)}{k} \tag{15}$$

where v is a node in the Markov chain, in this case $v \in M$. $C(v)$ is the total of the edge weights for all edges between node v and some other node, and $k = \sum_{v \in V} C(v)$.

Equation 15 can be used to find the distribution over the nodes in the Markov chain model of the hierarchy. This equation gives the distribution over all nodes, not just the primitive nodes. So, after extracting the probabilities of the primitive action nodes from the full distribution, the probabilities must be re-normalized. This is shown in Eq. 16 where ω^* is the stationary distribution over only primitive subtask nodes.

$$\omega^*(a|s) = \frac{C(a|s)}{\sum_{x \in A} C(x|s)},\ \forall a \in A \subseteq M. \tag{16}$$

In this equation, a is a primitive action node, A is the set of all primitive action nodes and M is the set of all subtask nodes. $C(a)$ is the sum of all of the weights of all of the edges

[4] Most task hierarchies, being poly-trees will be bipartite. However, it is possible to imagine non-bipartite hierarchies. For example, consider a hierarchy with a root node that has two children: a composite task A and a primitive task b. If A also has b as a child, then the graph is non-bipartite.

entering node a. The distribution needs to be conditioned on the state s because different states have different sets of subtasks terminated. The edge of the weights in this Markov chain was defined in Eq. 12. Plugging this into Eq. 16 gives Eq. 17.

Let $par(a)$ be the parent of node a and $child(M_i, s)$ be the number of children of subtask M_i available in state s.

$$\omega^*(a|s) = \frac{\sum_{M_a \in par(a)} \frac{1}{|child(M_a,s)|+1}}{\sum_{x \in A} \sum_{M_x \in par(x)} \frac{1}{|child(M_x,s)|+1}}. \tag{17}$$

The probability of a state s under weakly polled sampling, $P_{WP}(s)$, is as follows: $P_{WP}(s) = \sum_c P^*_{WP}((c, s))$, where $P^*_{WP}((c, s))$ is the stationary distribution over the hierarchical state space while running weakly polled sampling.

The probability of a state s using uniform random sapling is: $P(s) = \sum_c P^*((c, s))$, where $P^*((c, s))$ is the stationary distribution over the hierarchical state space using uniform random sampling.

Therefore, the KL-divergence of the weakly polled distribution with respect to the target distribution can be written as Eq. 18.

$$\begin{aligned} D_{KL}(P||P_{WP}) &= \sum_{s \in S} \sum_{a \in A} P(a|s)P(s) \ln \frac{P(a|s)P(s)}{\omega^*(a|s)P_{WP}(s)} \\ &= \sum_{s \in S} \sum_{a \in A} P(a|s)P(s) \\ &\quad \ln \left[\frac{P(a|s)P(s) \sum_{x \in A} \sum_{M_a \in par(x)} \frac{1}{|child(M_x, s)| + 1}}{\sum_{M_a \in par(a)} \frac{1}{|child(M_a, s)| + 1} P_{WP}(s)} \right]. \end{aligned} \tag{18}$$

In the bipartite case the action nodes can be divided into the two bipartite classes: A_{even} and A_{odd}. Then distributions for each class can be calculated as follows.

$$\begin{aligned} &\omega^*_{even}(a \in A_{even}|s) \\ &\quad = \frac{\sum_{M_a \in par(a)} \frac{1}{|child(M_a, s)| + 1}}{\sum_{x \in A_{even}} \sum_{M_x \in par(x)} \frac{1}{|child(M_x, s)| + 1}} \\ &\omega^*_{odd}(a \in A_{odd}|s) \\ &\quad = \frac{\sum_{M_a \in par(a)} \frac{1}{|child(M_a, s)| + 1}}{\sum_{x \in A_{odd}} \sum_{M_x \in par(x)} \frac{1}{|child(M_x, s)| + 1}}. \end{aligned}$$

The KL-divergence terms for each set of distributions can be written in the same manner as for the non-bipartite case. Then the two error terms can be added together. Therefore the total KL-Divergence is just the summation of these two terms over all states and actions as shown in Eq. 19.

$$D_{KL}(P||P_{WP}) = \left[\sum_{s\in S}\sum_{a\in A_{even}} P(a|s)P(s) \ln \frac{P(a|s)P(s)\sum_{x\in A_{even}}\sum_{M_a\in par(x)} \frac{1}{|child(M_x,s)|+1}}{\sum_{M_a\in par(a)} \frac{1}{|child(M_a,s)|+1} P_{WP}(s)}\right] + \left[\sum_{s\in S}\sum_{a\in A_{odd}} P(a|s)P(s) \ln \frac{P(a|s)P(s)\sum_{x\in A_{odd}}\sum_{M_a\in par(x)} \frac{1}{|child(M_x,s)|+1}}{\sum_{M_a\in par(a)} \frac{1}{|child(M_a,s)|+1} P_{WP}(s)}\right] \quad (19)$$

The weakly polled distribution $\omega(a|s)$ is the uniform random distribution $P(a|s)$, but with the number of sibling actions increased by one for each subtask. So for a given hierarchy structure, the new weakly polled distribution is guaranteed to be different, and therefore the KL-divergence is guaranteed to be nonzero. □

Appendix B: domain hierarchies

This appendix shows the task hierarchies for the domains used in the experiments. Figure 9 shows the task hierarchy for the Resource-Collection domain. Figure 10 shows the task hierarchy for the Parallel Resource Collection domain. Figure 11 shows the hierarchy for the Pseudo-rewards Resource Collection domain.

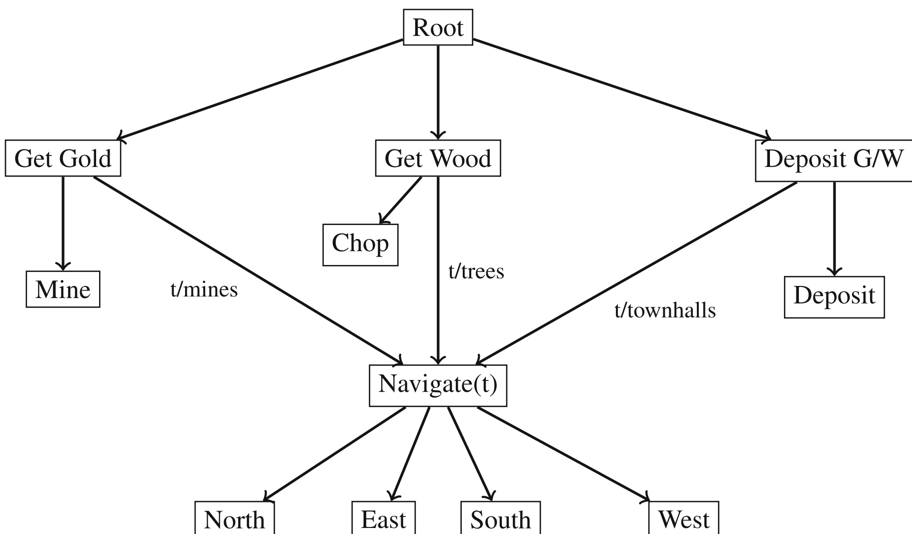

Fig. 9 Resource-Collection hierarchy

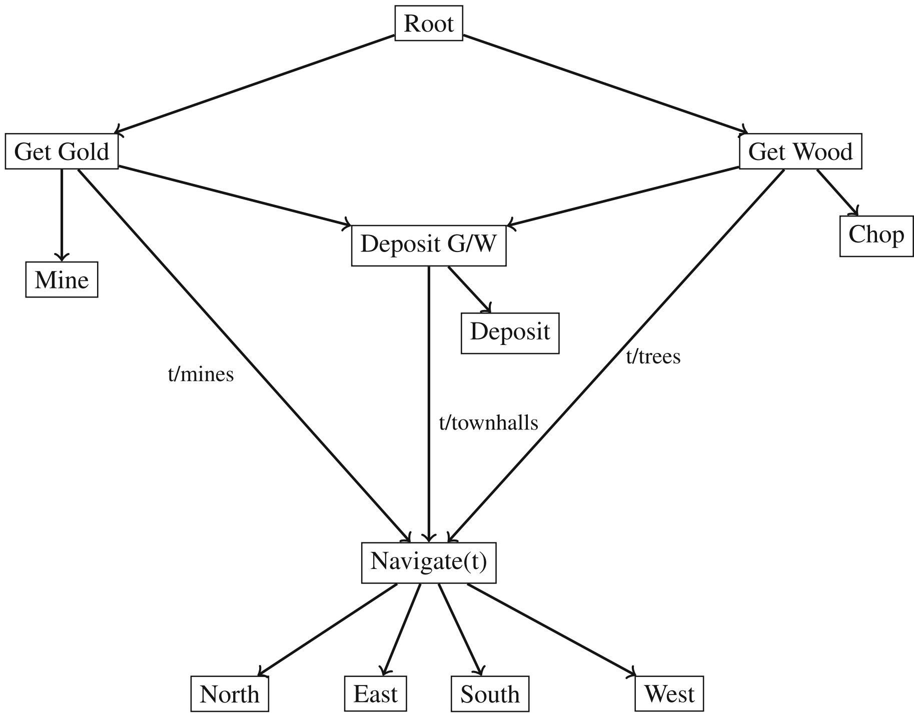

Fig. 10 ParallelRC

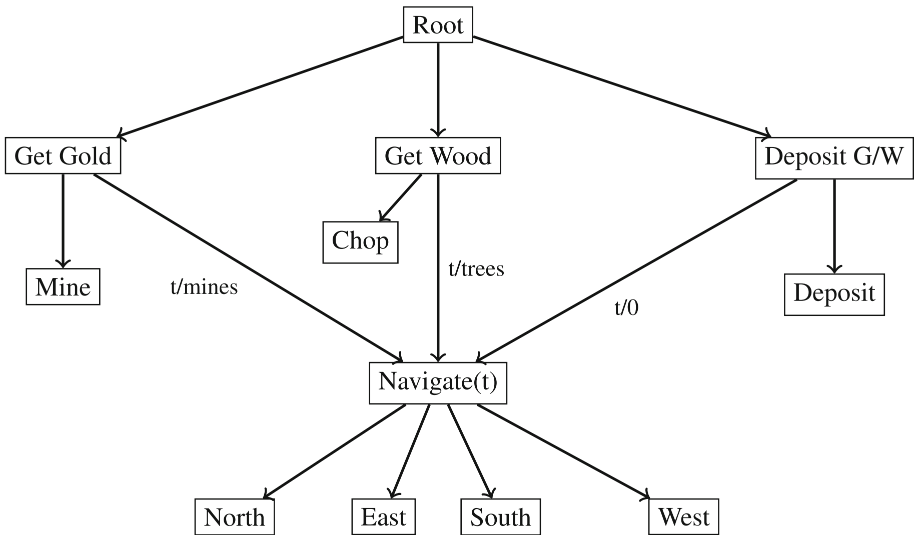

Fig. 11 PseudoRC hierarchy. *Deposit G/W* binds Navigate's t parameter to 0. When $t = 0$ navigate will be terminated whenever the agent is near any townhall, rather than a specific townhall.

Appendix C: raw data

Figures 12 and 13 show the average cumulative rewards for each of the experiments presented in the main paper.

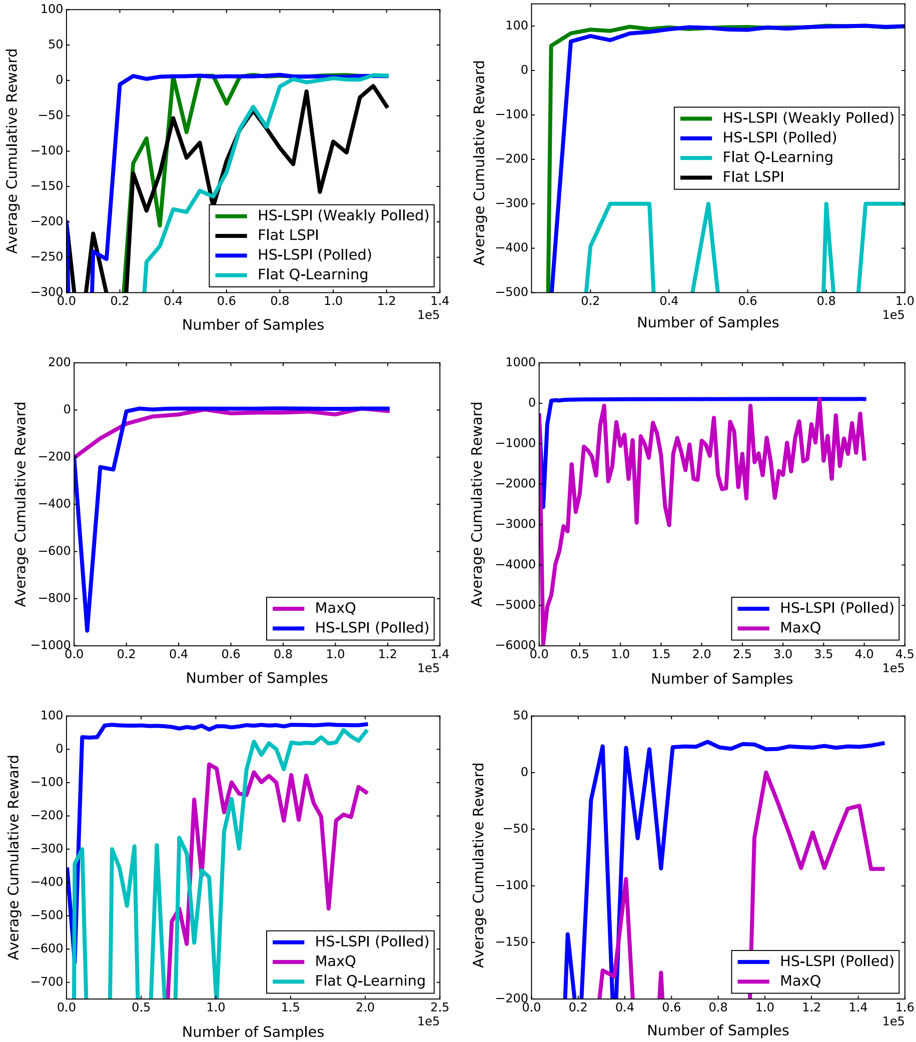

Fig. 12 Cumulative reward of the updated policy at each step. *Top row* Standard LSPI and Q-Learning versus hierarchical sampling LSPI methods on Taxi-World (*left*) and Resource-Collection (*right*). *Middle row* MAXQ versus hierarchical sampling LSPI methods on Taxi-World (*left*) and Resource-Collection (*right*). *Bottom row* MAXQ versus hierarchical sampling LSPI methods on Parallel-RC (*left*) and Pseudorewards-RC (*right*). The x-axis is Number of Samples and y-axis is average cumulative reward in all cases.

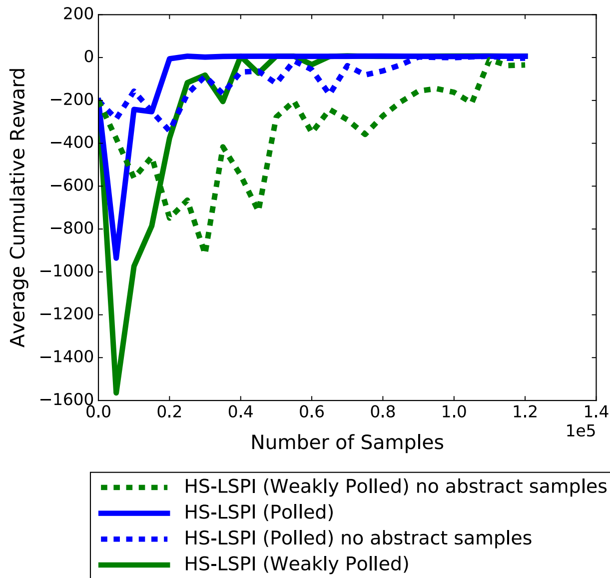

Fig. 13 Cumulative reward of the updated policy at each step. HS-LSPI polled sampling with inhibited action samples versus HS-LSPI polled sampling with inhibited action *and* abstract samples on Taxi-World .

References

Abbeel, P., Coates, A., & Ng, A. Y. (2010). Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, *29*(13), 1608–1639.

Aldous, D., & Fill, J. (2014). *Reversible Markov chains and random walks on graphs*. Berkeley: University of California Berkeley.

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Association for the advancement of artificial intelligence/innovative applications of artificial intelligence conference* (pp. 119–125).

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*(4), 341–379.

Cao, F., & Ray, S. (2012). Bayesian hierarchical reinforcement learning. In *Advances in neural information processing systems*.

Comanici, G., & Precup, D. (2010). Optimal policy switching algorithms for reinforcement learning. In *Proceedings of the 9th international conference on autonomous agents and multiagent systems* (Vol. 1, pp. 709–714). International Foundation for Autonomous Agents and Multiagent Systems.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Jonsson, A. (2006). Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research*, *7*, 2259–2301.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., & Tenenbaum, J. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems* (Vol. 29, pp. 3675–3683).

Lagoudakis, M. G., & Parr, R. (2003). Least-Squares Policy Iteration. *Journal of Machine Learning Research*, *4*, 1107–1149.

Lazaric, A., Ghavamzadeh, M., & Munos, R. (2012). Finite-sample analysis of least-squares policy iteration. *Journal of Machine Learning Research*, *13*, 3041–3074.

Lovász, L. (1996). Random walks on graphs: A survey. In D. Miklós, V. T. Sos, & T. Szonyin (Eds.), *Combinatorics: Paul Erdős is eighty, Bolyai society mathematical studies* (pp. 353–348). Budapest: János Bolyai Mathematical Society.

Marthi, B., Russell, S., & Latham, D. (2005). Writing stratagus-playing agents in concurrent ALISP. In *Reasoning: Representation, and learning in computer games* (p. 67).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.
Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*.
Puterman, M. L. (2005). *Markov decision processes: Discrete stochastic dynamic programming*. Hoboken, NJ: Wiley.
Schultink, E., Cavallo, R., & Parkes, D. C. (2008). Economic hierarchical q-learning. In *Proceedings of the 23rd national conference on artificial intelligence*.
Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.
Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation* (pp. 212–223). Springer.
Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1–2), 181–211.
van Seijen, H., & Sutton, R. (2015). A deeper look at planning as learning from replay. In *International conference on machine learning* (pp. 2314–2322).
Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292.
Wiering, M., & van Otterlo, M. (2012). *Reinforcement learning state-of-the-art*. Berlin: Springer.
Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *International joint conference on artificial intelligence* (Vol. 95).