Hierarchical Explanation-Based Reinforcement Learning

Prasad Tadepalli and Thomas G. Dietterich
Computer Science Department
Oregon State University
Corvallis,Oregon 97331-3202
{tadepalli,tgd}@research.cs.orst.edu

Abstract

Explanation-Based Reinforcement Learning (EBRL) was introduced by Dietterich and Flann as a way of combining the ability of Reinforcement Learning (RL) to learn optimal plans with the generalization ability of Explanation-Based Learning (EBL) (Dietterich & Flann, 1995). We extend this work to domains where the agent must order and achieve a sequence of subgoals in an optimal fashion. Hierarchical EBRL can effectively learn *optimal* policies in some of these sequential task domains even when the subgoals weakly interact with each other. We also show that when a planner that can achieve the individual subgoals is available, our method converges even faster.

1 Introduction

Reinforcement Learning (RL) has emerged as the method of choice for building autonomous agents that improve their performance with experience. One obstacle to scaling this approach to large problems is the lack of a robust and justifiable method to generalize from one experience to another. Dietterich and Flann (Dietterich & Flann, 1995) showed that Explanation-Based Learning (EBL) can be used to generalize the experience of a reinforcement learner across different states, provided the learner has access to a complete and correct domain theory. The result is an effective learning method called Explanation-Based Reinforcement Learning (EBRL) that enjoys the advantages of both RL and EBL. Like RL, but unlike pure EBL, it can learn optimal plans. Like EBL and unlike table-based RL, it can also effectively generalize its experience to other similar states in a manner justified by the domain theory.

Dietterich and Flann illustrated the use of EBRL in a maze task and in chess endgames. Both of these tasks have goals that cannot be decomposed into a sequence of subgoals. However, in many real-world domains, the agents must achieve multiple subgoals sequentially. The agents may order the subgoals to minimize the cost (or maximize the reward) of achieving them. Although there may be some degree of independence among the subgoals, there may also be weak interactions between them. Applying EBRL to these domains straightforwardly does not give any advantage, because the complexity of the sequence of subgoals is such that the optimal action sequence would almost never be the same for two different states. Hence, EBRL will not be able to generalize the value functions across different states.

In this paper, we illustrate a method that learns effectively in such domains. Our approach is to introduce a two-level hierarchy where the subgoals are ordered at the top-level and are solved in the lower level. When the subgoals at the lower level have only a few interactions, optimal ways to solve them can be effectively generalized and compactly represented using EBRL. The EBL component of EBRL keeps track of the features which are relevant for a sequence of moves to be successful in achieving the subgoal at the lower level and learns a value function which is represented as a list of pattern-value rules.

We empirically show that in some domains, the value function at the higher level is small enough that it can be easily represented as a table. Unlike some other research in the RL literature where optimality is sacrificed for efficiency by using a hierarchy (Dayan & Hinton, 1993; Kaelbling, 1993), in our approach, optimality is retained. In this respect, it is similar to (Singh, 1992), although, unlike in Singh's work, here we do not require the system to achieve the subgoals in a fixed order to get the reward. The subgoal order can be decided by the system based on the considerations of subgoal interactions and plan optimality. Unlike these earlier studies, where the role of the hierarchy is to enable the reuse of subgoals, our method uses a hierarchy primarily to achieve the *partitioning of concerns*: we exploit the near independence of optimally solving each subgoal from optimally ordering the subgoals. This yields a huge reduction in the complexity of the value function.

We show that hierarchical EBRL is significantly faster than undiscounted Q-learning (which optimizes total reward) and R-learning (which optimizes average reward). In both cases, we use a form of "optimism under uncertainty" for exploration (Kaelbling, Littman, & Moore, 1996; Koenig & Simmons, 1996; Ok & Tadepalli, 1996).

In many domains, a planner is available that can achieve the low level subgoals in an optimal fashion. We experimented with a means-ends-analysis planner for exploration at the lower level of the hierarchy, while using optimism under uncertainty at the higher level. The results here confirm our intuition that the presence of a planner reduces the burden on the learner and makes it converge to an optimal policy faster.

The rest of the paper is organized as follows. In the next section, we give a brief overview of undiscounted Q-learning, EBL, and EBRL. In Section 3, we introduce hierarchical EBRL and discuss how it can take advantage of a planner. In Section 4, we present empirical results that support our claims. In Section 5, we discuss future work and conclude.

2 Background

We assume the standard framework of Markov Decision Processes (MDP), which are described by a set of n discrete states S and a set of actions A. The actions that are applicable in a state i are denoted by $U(i)$ and are called *admissible*. In this paper, we assume a deterministic universe in which there is a unique next state $Next(i,u)$ for every state i and every action u. There is a finite immediate reward $r(i,u)$ for executing an action u in state i. A state is absorbing if every action executed in that state keeps the system in the same state. We assume that there is a set of absorbing goal states in S.

Time is treated as a sequence of discrete steps $t = 0, 1, 2, \ldots$. A *policy* μ is a mapping from states to actions such that $\mu(i) \in U(i)$. We only consider policies that do not change with time (i.e., stationary policies). A trial consists of starting from some randomly chosen start state and reaching an absorbing goal state. For now, assume that the goal is to maximize the expected total reward received before reaching an absorbing goal state.

One reinforcement learning approach for solving this problem is undiscounted Q-learning (Watkins & Dayan, 1992). This method learns by updating the value of $Q(i,u)$ for each $i \in S$ and $u \in A$, which represents the expected total reward of executing action u in state i and from then on following the optimal policy. Hence, the correct Q values must satisfy the following relationship, where $U_Q(j) = \max_a Q(j,a)$ is the "utility" or the expected total reward for starting in state j and following the optimal policy:

$$Q(i,u) = r(i,u) + U_Q(Next(i,u)). \qquad (1)$$

The Q-values may be learned by updating the $Q(i,u)$ with the right hand side of Equation 1, whenever an action u was executed in state i. If the Q-values of a state i are known, then the optimal action a in that state can be easily found to be the one that maximizes $Q(i,a)$. Similarly, if we know the utilities $U_Q(j)$ of all states j and the action and reward models, we can also compute the optimal action by comparing the Q-values computed for various actions from Equation 1.

One problem with the above method, however, is that it is table-based—the Q values must be maintained for every state-action pair. However, in many domains which have intensionally described parameterized actions, a large number of states can reach the goal state using the same abstract action sequence.[1] All of these states will have a value which is at least as high as the effective reward of this action sequence. Using the action and reward models ($Next(i,u)$ and $r(i,u)$ above), which are now represented intensionally, EBL can derive an abstract description of the states — a "pattern" which describes a "region" of the state space — for which the same action sequence leads to the goal.

EBL does not necessarily result in an optimal policy, however, because it keeps track of only the first path found from any state rather than the best possible path. EBRL corrects this problem by associating with each pattern the utilities of the states that match that pattern. If a state matches two patterns, then its utility is the higher of the utilities of the two patterns (Dietterich & Flann, 1995). The EBRL method directly learns U_Q-values for patterns of states rather than Q-values. Dietterich and Flann showed that it is more effective than either pure RL or pure EBL in converging to the optimal policy in a maze task and in rook and knight endgames in chess.

[1]This is true when each action has a large number of "irrelevant" features of the state that it does not effect, and is not true for a maze-like task, where there are no irrelevant features. For example, if location is the only feature of the state and is affected by every operator, no generalization may be expected to be produced by EBL.

3 The Approach

Reinforcement Learning as introduced so far requires re-learning of the value function whenever the task is changed. This is true with or without the EBL component. One way to accommodate changing tasks in this paradigm is to treat the task as part of the state. This, however, has the effect of increasing the size of the search space and fails to take advantage of the structure of the tasks, which is sometimes readily available.

3.1 Sequential Task Domains

In many real-world domains, the agent is often required to perform multiple tasks. However, the physics of the world and the agent impose a sequential constraint on the tasks, in that the agent has to perform them one after another. The tasks may have mutual interactions so that some orderings of tasks are not feasible. Correctly ordering the tasks is often crucial for optimal performance. In many cases, the reward may be delayed until the agent completes all the tasks. We call these domains "sequential task domains." Examples of this kind of domain include delivering mail, transporting jobs from one machine to another on the factory floor, and many other versions of the standard traveling salesperson problem. Even serializable planning problems such as mechanical assembly, where agent location is not an important aspect of the state, can be put in this framework.

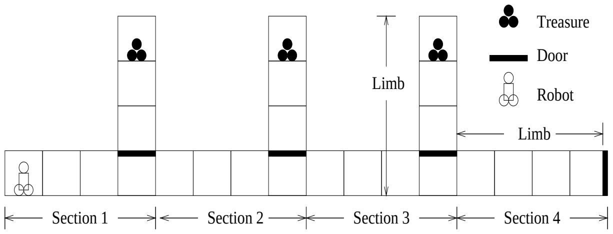

Figure 1: The treasure-hunt domain with 4 sections and a limb size of 4. The goal is to collect all the treasures and exit through the rightmost door.

In this paper, we use a toy domain, which we call the "treasure-hunt domain," that has all of the above characteristics. As shown in Figure 1, this domain consists of a set of vertical corridors joined by a horizontal corridor. The vertical corridors lead to cells that may contain treasures. There is a door at the entrance of each of the vertical corridors that must be opened to enter it. The task for the robot is to collect all the treasures available as fast as possible and exit through the rightmost door. A new trial starts as soon as the robot exits this door. Each vertical corridor, and the part of the horizontal corridor between two vertical corridors is called a "limb." The total number of horizontal limbs is called the number of "sections." The task in each trial may be slightly different, depending on which corridors have treasures in them and which doors are open or closed. To make the goals interact in an interesting way, we require that all other doors must be closed before the final door at the right can be opened. A positive reward is given immediately after each treasure is acquired.

3.2 Hierarchical EBRL

In many sequential task domains, including the treasure-hunt domain, it is possible to decompose the world state into two components: the location of the robot and the states of all objects of interest in the domain (including the robot itself). Every object has a specific location, and the robot has to be in that location to change the object's state. Moreover, there is a finite limit on the number of objects in any one location (usually 1, or 2 including the robot). Let us also assume that there is no reward for just being in a particular location. This is not a restrictive assumption in that if there is a location in which we prefer the robot to be, we can imagine a virtual "object" in that location that changes its state depending on whether the robot is there. Consider a sequence of actions taken by such a robot following an optimal policy in such a domain. Since the robot can only be in one location at a time and it can only change the states of the objects in its location, the robot's actions will necessarily be of the form: go to location L_1, change the states of objects in L_1, go to location L_2, change the states of objects in L_2, and so on.

Hence there are two problems to be solved to find the optimal policy: (a) finding the best sequence of actions that achieves a required change in the state of an object, and (b) determining the optimal sequence of object state changes. We capture this intuition by decomposing the value function (e.g., Q-value) into two parts: the D-function will estimate the maximum total reward that can be achieved by starting with a given action and changing the state of an object. The G-function estimates the maximum total reward that is achievable from that point forward *after* the state of the object is changed. The following equation decomposes the Q-values for state-action pairs:

$$\begin{aligned} Q(state, action) \;=\; \max_{sgoal}\, \{&D(state, sgoal, action) \\ &+G(state_2)\ \} \end{aligned}$$

where, $sgoal$ refers to the subgoal of changing the state of some object to some value, and $state_2$ refers to the state after the subgoal has been achieved. Note that we are assuming that in achieving $sgoal$, we only change the state of the object involved and the location of the robot. In general, this kind of decomposition is possible if there are no remote-control actions, i.e., the robot should be next to the object to change its state, and no action changes the states of two or more objects. We can use a similar decomposition for the U_Q-values of states as well.

On the face of it, approximating the two functions D and G might appear more complex than approximating just the Q-function. Two observations suggest why this may not be the case. First, since the robot is always at the location of the object when its state is changed, in computing the G-function, the location of the robot is limited to the locations of one of the objects. If the number of objects is much smaller than the number of possible locations of the robot, as is usually the case, the G-function can be much more compact than the Q function. Second, the value estimated by the D-function might be independent of most of the features of the state, because it estimates only the reward obtained in changing the state of a single object. In an ideal case, it may only be a function of the location of the robot and the object whose state is changed. If there are subgoal interactions, however, its value might also depend on the other objects in the state. For example, in the treasure-hunt domain, the reward received in getting the treasure might also depend on the state of the door leading to the treasure. It is these kinds of dependencies that the explanation-based component of the system can extract from the sequence of moves made and their preconditions. Once these dependencies are explicated, the system knows all the features relevant for a state to have a certain D-value, which allows it to create a rule that assigns the same D-value to all states that satisfy the rule.

We can do a rough analysis of the space savings due to this decomposition. Let the number of possible locations of the robot be l. Let $|A|$ be the number of possible actions, and n be the number of objects in the domain. Further, assume that to change the state of any single object, the only features relevant are the location of the robot and the states of at most r objects. The number of world states $|S| = 2^n l$, and hence the number of possible entries in the Q-table is $2^n l|A|$. On the other hand, since the number of relevant features to consider in achieving a single subgoal is bounded by r, the number of rules needed to represent the D-function is bounded by $2^r nl|A|$. If $r << n$, this is going to be much smaller than the size of the Q-table. Since the robot must be at one of n possible locations immediately after changing the state of an object, the G table needs to be no larger than $n2^n$. If $n << l$, once again, this is much smaller than the size of the Q-table. This suggests that the ideal condition for superior performance of Hierarchical EBRL over Q-learning is $r << n << l$. The space savings that occurs when this condition holds translates directly to increased speed of convergence since the number of rules or table entries to be learned is reduced.

3.3 Updating the value functions

In this section, we describe the details of how the G and D functions are represented and updated in our system.

Recall that the G-value is relevant only for those states in which the robot is in the same location as some object. Hence, we represent the G-value of such world states with a table, which is indexed by the states of all objects in the world and the index of the object that the robot is at.

Representing the D-function is more complicated, because it is potentially a function of different features in different regions of the state space. We use a representation, which we call the "value list," that consists of a list of clauses. Each clause assigns a value to a set of states that satisfy a conjunctive condition. The semantics of the value list is like a COND-clause in Lisp; the value of a state is the value of the first clause whose condition is satisfied by the state. We have one value clause for each (subgoal, action) pair.

Updating the D-table is similar to updating the Q-value (see Equation 1), except that we do the updates for each possible subgoal in each step, and we do a symbolic propagation of the clauses as well as numeric propagation of values. More formally, suppose that an action u was executed in state i resulting in state j. Let the value list for a subgoal g for action a_k be:

if $c_{k,1}$ **then** $v_{k,1}$**, else if** $c_{k,2}$ **then** $v_{k,2}$**,** $\ldots$**, else if** $c_{k,n}$ **then** $v_{k,n}$

For each action a_k, let c_{k,m_k} be the first condition that matches the state j. The utility of the state j, $U_Q(j)$, must be the maximum of all the corresponding values over all actions, i.e., $\max_k v_{k,m_k}$. This is added to the immediate reward $r(i,u)$ to give the backed up value for the state-action pair (i,u).

We also want to generalize this to other "similar state-action" pairs (i',u), which are those in which the same action leads to a state j' similar to j, i.e., a state that has the same best value and best action. This means that j' must satisfy the conditions $F_{pos} = \{c_{k,m_k} | 1 \leq k \leq n\}$.

Unfortunately this is not enough. If a_b is the best action in state j, since the clauses in the value list are matched sequentially, j' must not match any clause that occurs before c_{b,m_b}, since otherwise its U_Q value will be different from that of j. Similarly, it should also not match a condition $c_{k,p}$ that occurs before c_{k,m_k} for any action $k \neq b$ if $v_{k,p} \geq v_{b,m_b}$, because otherwise a_b would not be the best action in that state.

The above implies that we must ensure that the result of action u does *not* satisfy a set of conditions: $\{c_{b,p} | p < m_b\} \cup \{c_{k,p} | k \neq b, p < m_k, v_{k,p} \geq v_{b,m_b}\}$.

Since our value list language does not allow negations, we look for the features of state j that are sufficient to make all the above conditions false.[2] To maximize the generality of the resulting pattern, we use the greedy set cover algorithm to find an approximately minimal set of features. To this set of features F_{neg}, we add the features F_{pos}, and regress the union over the description of action u, i.e., subtract the features changed by action u and add the features in the preconditions of action u. This gives the set of features of state i so that choosing action u in any state i' satisfying these features will lead to a state j' similar to j.

We thus create a new value clause "If Regress($F_{neg} \bigcup F_{Pos}, u$) then $r_{imm} + v_{b,m_b}$," and add it to the front of the value list for action u and the subgoal g.[3]

For example, let the doors in our treasure-hunt domain of Figure 1 be labeled d_1 through d_4 from left to right, and let the X-coordinates of the grid cells be labeled 1 through 16 also from left to right. Assume that the robot has just executed the action Right from the cell whose X-coordinate is 15, reaching the rightmost cell, and making the predicate $X-loc(16)$ true. Let us call the first state of the robot i and the second state j. Now, $D(obj,\text{Right})$ will be updated for each object obj. Consider the case where $obj = d_4$, the last door. Suppose that the door d_3 is open and the other doors are closed in state j, and that the D-value lists stored for Open and Left for d_3 are, respectively:

Open: **if** $Xloc(16) \wedge closed(d_1) \wedge closed(d_2) \wedge closed(d_3) \wedge closed(d_4)$ then -1,
else if $Xloc(16) \wedge closed(d_4)$ then -30.

Left: **if** $Xloc(16) \wedge closed(d_1) \wedge closed(d_2) \wedge closed(d_3) \wedge closed(d_4)$ then -3,
else if $Xloc(16) \wedge closed(d_4)$ then -20.

Since d_3 is open, state j matches only the second conditions of the value list for both the actions, of which Left has a better value. Hence, $F_{pos} = \{Xloc(16), closed(d_4)\}$. The generalization should not match the first clauses of either action, or else, either the best value or the best action for state j would be different. Hence, we use greedy set cover to pick the features of state j, which, while being consistent with features in F_{pos}, negate at least one of the conditions in the first clauses of the two actions. Assuming that greedy set cover returns $F_{neg} = \{open(d_3)\}$, $F_{pos} \cup F_{neg}$ is regressed over the action Right, yielding the condition $\{Xloc(15), closed(d_4), open(d_3)\}$. If the action Right has a penalty of -1, then this condition is associated with the value $-1-20=-21$, and is made a new value clause for the pair (d_4, Right). This rule says that if you are at a location whose X-coordinate is 15, and d_4 is closed and d_3 is open, the utility of moving right is -21. (The utility is low because d_3 must be closed before d_4 can be opened.) Notice that this rule mentions only d_3 and not d_1 or d_2 because d_3 is the door which is open in the current example and led to the violation of the two first clauses. It also does not mention the states of any treasures, because they are irrelevant to the action and not mentioned in the relevant value lists of the next state.

[2]There are such features in j since otherwise j itself would have a different value or a different best action.

[3]In the stochastic case, we combine this value with any existing value clause with the same condition using the stochastic update rule of Q-learning.

Updating the G-function is straightforward, since it is implemented as a table, indexed by the object at which the agent currently is, and the states of all objects in the current state. In particular, we use the following update formula whenever an object o_1 changes its state. In the following equation, $state_1$ and $state_2$ are world states immediately after the objects o_1 and o_2 changed their states.

$$G(state_1) \leftarrow \max_{o_2,act}\{D(state_1, o_2, act) + G(state_2)\} \tag{2}$$

The left hand side represents the total reward one can gain from $state_1$, which is the state immediately after changing the state of object o_1. The right hand side of Equation 2 corresponds to the sum of the reward gained up to and including changing the state of the object o_2 starting from $state_1$, and the total reward gained after that until the end. The max operation maximizes over all possible objects that can be picked as o_2 and all possible first actions, act, that can be executed in $state_1$.

3.4 Exploration and Planning in Hierarchical EBRL

We know that without sufficient exploration of the state space, RL methods run the risk of converging to a locally optimal policy. One way to encourage exploration is to initialize the value function over states (or state-action pairs) with high enough values and give a constant penalty for each action so that the learner believes that the states that have not been sufficiently visited have a high value and tends to visit them. This is called "optimism under uncertainty" and is found to be an effective exploration technique in many domains (Kaelbling et al., 1996; Koenig & Simmons, 1996; Ok & Tadepalli, 1996).

In Hierarchical-EBRL, we can implement optimism under uncertainty by initializing both the D-function and the G-function with 0's, and by giving a constant penalty for each action and a positive reward for goal actions, e.g., picking up treasure in the treasure-hunt domain. In any state $state$, the system simply picks the best subgoal and action pair using Equation 3, where $state_2$ is the world state after the object obj changes its state.

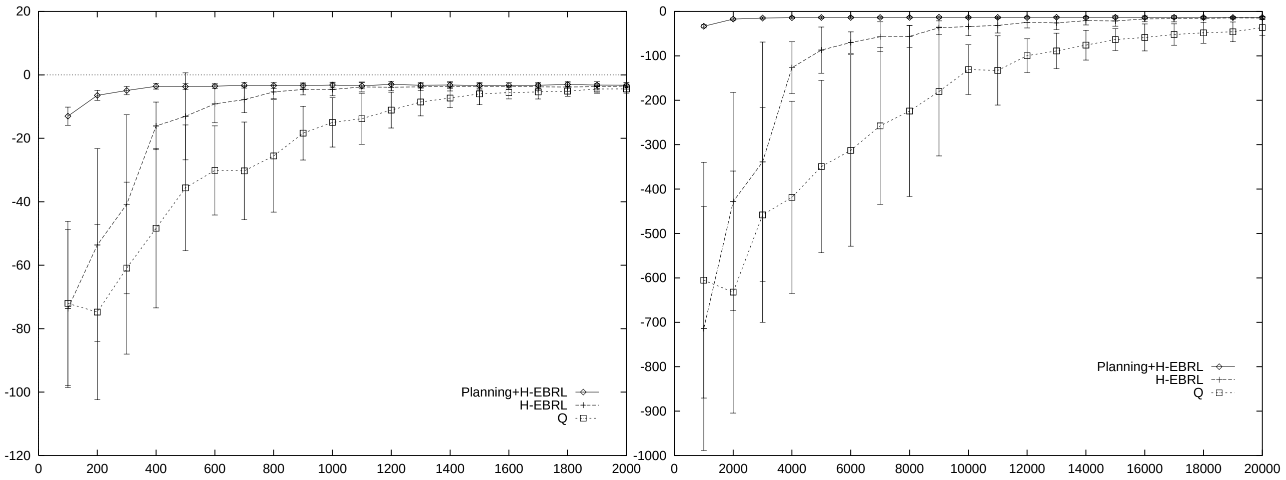

Figure 2: Total reward per trial vs. the number of steps. (a) limb size = 2, sections = 3 (left), and (b) limb size = 10, sections = 3 (right).

$$arg \max_{obj,act} \{D(state, obj, act) + G(state_2)\} \qquad (3)$$

In many domains, it is possible to plan for single subgoals efficiently with an adequate domain theory. In our treasure-hunt domain, for example, it is easy to plan to close a single door or to pick up a single treasure. It is planning the best possible way to combine the solution of several subgoals that is complex. Under these conditions, it makes sense to ask whether a local planner that knows how to solve subgoals can improve the speed of convergence of Hierarchical EBRL. Another way of asking the same question is how much of the exploration that the Hierarchical EBRL does is to learn about solving single subgoals, and how much of it is to learn optimal subgoal ordering.

It is fairly straightforward to combine a subgoal planner with Hierarchical EBRL. There is one problem, however. Using a planner at the low level is not compatible with optimism under uncertainty. Optimism under uncertainty encourages exploration so that the best possible plan is eventually found. If the planner is already optimal at the level of planning for subgoals, there is no point in exploring at that level. In fact, optimism under uncertainty at the subgoal level completely negates the advantage gained by having a planner. Hence, when there is an optimal planner available for solving subgoals, the system should use it, assume that the resulting plans are optimal, and not explore alternatives. We call this "pessimism under uncertainty," because it believes that anything other than the planner's plan for solving a subgoal is going to be worse.

Pessimism under uncertainty is implemented in Hierarchical EBRL by initializing the D-function with a large negative value, and not updating the D-function after each action. The D-function is updated only after an entire subgoal is achieved (and in reverse execution order). We can still use optimism under uncertainty at the higher level of the hierarchy, since the system needs to explore to find the optimal subgoal order.

4 Experimental results

Our experiments are based on the treasure-hunt domain which we have designed to test the role of planning in Reinforcement Learning. There are 8 actions available to the agent: Up, Down, Left and Right are the standard navigation actions. Open and Close open and close doors, and Take picks up a treasure if the robot is in the same location as the treasure. A special action called Reset which is possible only after the last door is opened, simulates the starting of a new trial by randomly setting the states of doors and treasures and the location of the agent. The actions are admissible only in relevant locations, e.g., the action Open is available only at locations with doors. We give a penalty of -1 for each action and a small positive reward for taking the treasure, which is adjusted so that the best possible average reward per step is less than 0. This is to allow optimism under uncertainty, so that it is not possible to prematurely converge to a suboptimal policy with a small positive reward (since there is a penalty of -1 for each action).

We used optimism under uncertainty for our exploration, which means that the system always takes an action which is greedy as specified in Equation 3. When there is a planner available, we used pessimism

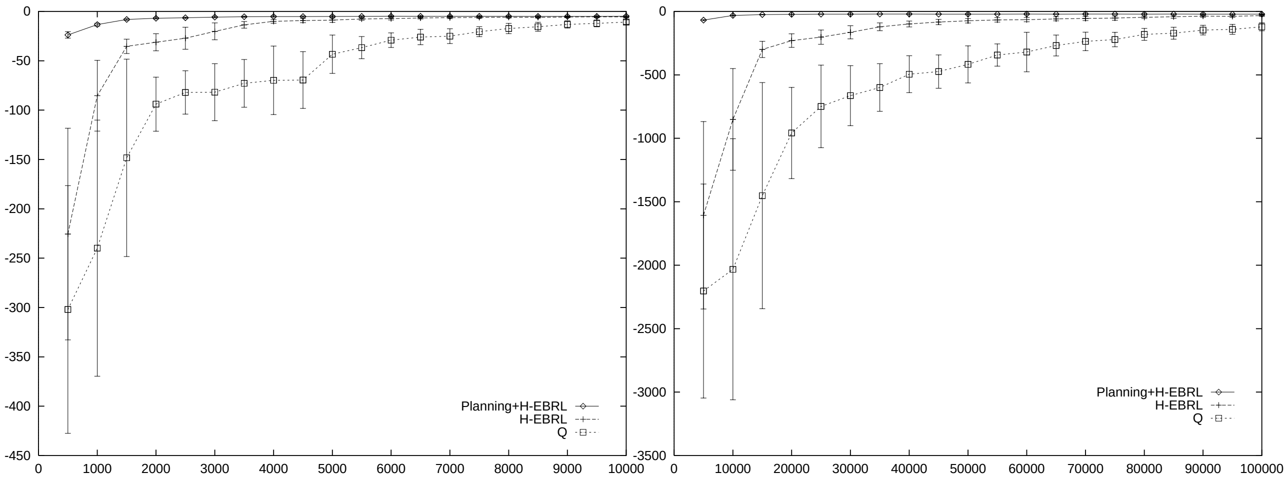

Figure 3: The average reward per step vs. the number of steps. (a) limb size = 2, sections = 4 (left), and (b) limb size = 10, sections = 4 (right).

under uncertainty at the lower level and optimism under uncertainty at the higher level. The actions are replayed backwards once the system successfully executes Reset. Each run consists of a long sequence of steps. Data was gathered for 50 runs for each experiment.

Figure 2 shows the results in the treasure-hunt domain with 3 sections and limb sizes 2 and 10. On the Y-axis is the total reward received per trial between two successive measurements for undiscounted Q-learning and Hierarchical EBRL with and without planning.[4] The graph clearly shows that Hierarchical-EBRL converges faster than Q-learning in both cases, and the advantage grows with the limb size.

Unfortunately, in this domain as in many sequential task domains, pure EBRL degenerates into Q-learning. This is because it is rare for two distinct states to have identical optimal action sequences for the whole trial. The optimal action sequence most often determines the states of all objects and the location of the robot. For example, executing the sequence of actions that constitute picking up the third treasure, closing the third door and opening the last door is optimal only when the robot is next to the third treasure, the other two treasures do not exist, and all doors except the third are closed. For this reason, we do not show pure EBRL separately from Q-learning in our results.

Figure 2 also shows how a planner can speed up the convergence even more in both cases. This is also a way of separating the effort involved in low-level and high-level exploration in Hierarchical EBRL. Another

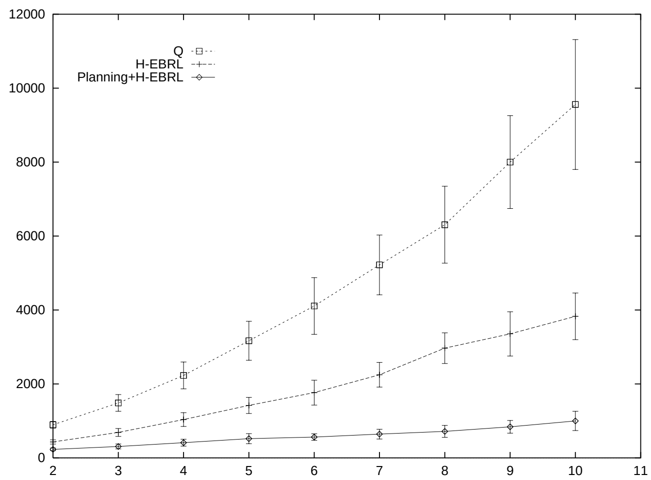

Figure 4: Growth of the number of steps for convergence with the limb size in the deterministic grid domain with fixed start state and 3 sections.

way of reading the results is that the difference between the EBRL with and without planning is the delay caused by not knowing the best way to solve a subgoal.

Figure 3 shows the results for 4 sections for limb sizes of 2 and 10. The results are similar to the case of 3 sections in that Hierarchical-EBRL is better than Q-learning; with planning it is even better. It also shows that the advantage improves with the limb size, as our analysis in Section 3.2 suggests, since longer limbs increase the number of values to learn in Q-learning much more than in Hierarchical-EBRL. Planning would exaggerate this difference even more.

[4]The error bars are 1 standard deviation apart on either side in all our graphs.

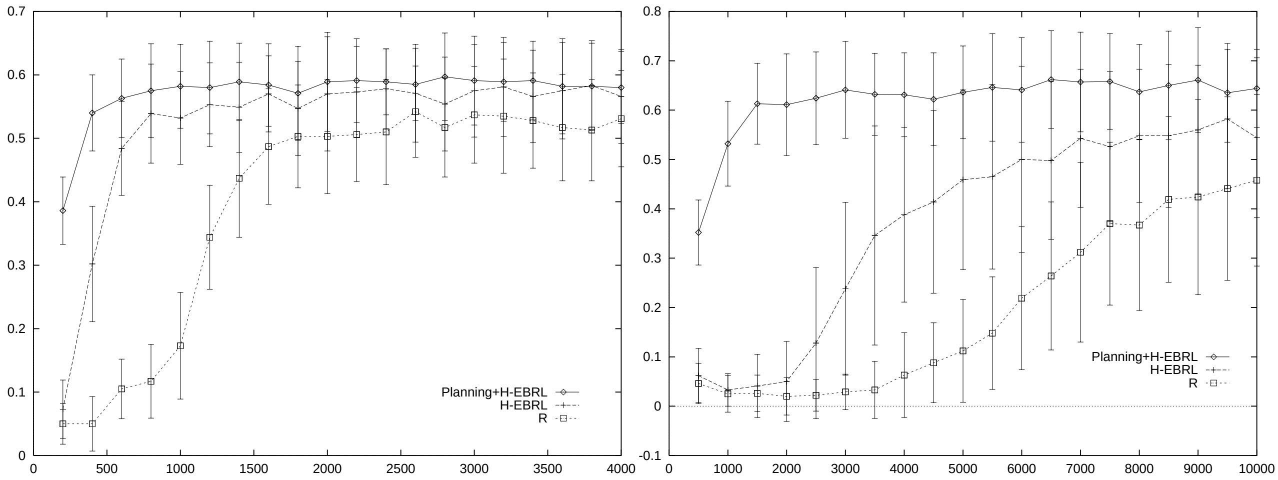

Figure 5: Average-reward per step vs. the number of steps for H-EBRL, EBRL, and R-learning (a) limb size = 2, sections = 3 (left), and (b) limb size = 10, sections = 3 (right).

Figure 4 summarizes the results in the simpler case, when the robot is always started in a fixed start state where all the treasures are present, all doors are closed and the robot is in the left-most cell. The number of sections was set to 3 and the limb size was varied from 2 to 10. The results show that number of steps for convergence varies at different rates for all 3 systems, planning with Hierarchical EBRL being the best and Hierarchical EBRL being the second best.

We also implemented a version of the Average-reward Hierarchical-EBRL method. This method uses a version of R-learning as the base method (Schwartz, 1993). It estimates the value of the average reward ρ of the current greedy policy and subtracts it from the immediate reward of each step before using it to update the D-function and the G-table. The values of the the states after the robot opens the last door are all forced to be 0's. This is justified because (a) only relative values are important for action selection, and (b) all these different states are "equivalent" in the sense that the probability of the next state for each action is the same for all of them. (See (Bertsekas, 1995) for a more rigorous explanation.) To encourage exploration, ρ is initialized to a high value and is slowly decreased with a learning rate α. This is the equivalent of optimism under uncertainty for Average-reward RL (Ok & Tadepalli, 1996). The initial values of ρ and α were adjusted by trial and error. Unlike in the total reward case, we did not use any penalty for actions, because the initial value of ρ plays the role of a penalty and encourages exploration. As before, there is positive reward for picking up each treasure.

Again, the results of Figures 5 and 6 suggest that Hierarchical-EBRL has a definite advantage over the base method, R-learning. Planning improves the performance of Hierarchical EBRL even more. The advantage grows with the domain size here as well.

5 Conclusions and Future Work

This paper shows that Hierarchical EBRL can be effective in learning optimal policies in sequential task domains. The main reason for the success of this method is the decomposition of the value function into simpler functions, which can be separately learned and combined at action selection time. Without a hierarchy, EBRL cannot succeed in this domain, because the sequential nature of the task prohibits the value function from generalizing across different states.

Almost all methods that employed hierarchies in RL before used them to enable subgoal reuse in essentially the same kind of task, namely navigation (Kaelbling, 1993; Dayan & Hinton, 1993; Singh, 1992). In contrast, the main source of power in our system is that it exploits the near independence of the navigation task from optimal planning or ordering of subgoals. This partitioning of concerns is very powerful in many real-world domains where each action affects only a small subset of the features of the world state. This makes it possible to optimize different subgoals separately by learning separate local evaluation functions and then combining the results. Moreover, when a correct domain theory that specifies the relations between the preconditions and the effects of actions is available, EBL can use it to extract more general mappings from states to values. It appears that the EBL component can be seen as a knowledge-based method for creating and refining a hierarchy, which may then be used for

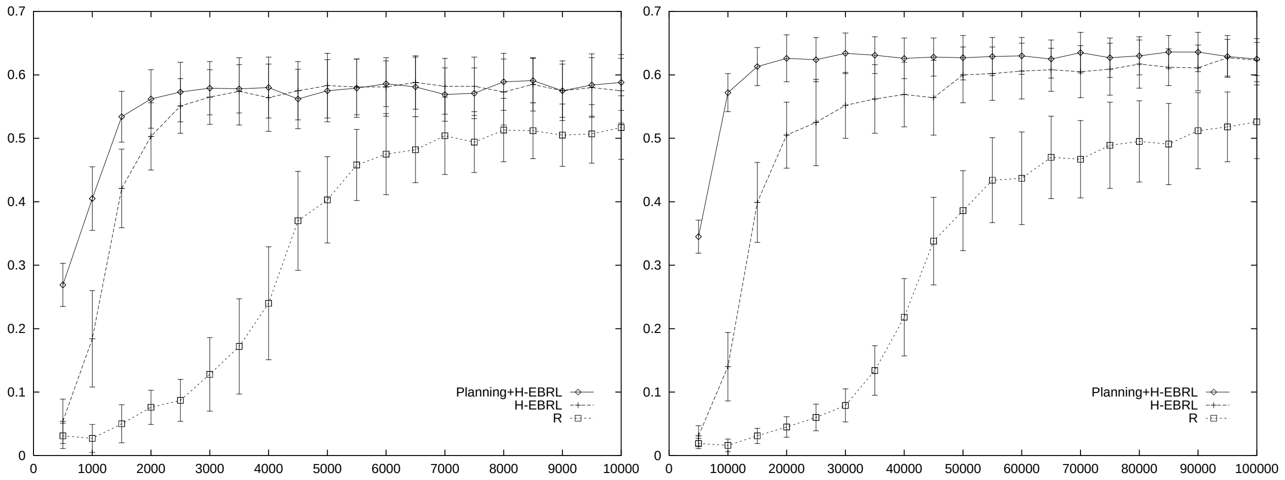

Figure 6: Average-reward per step vs. the number of steps for H-EBRL, EBRL, and R-learning (a) limb size = 2, sections = 4 (left), and (b) limb size = 10, sections = 4 (right).

efficient reinforcement learning.

Hierarchical EBRL inherits some problems of EBL such as the generalization-to-N problem, which is due to equating "an explanation" with an operator sequence (Shavlik, 1990). In our treasure hunt domain, this problem prohibits learning a value function that can generalize across different locations of the robot. It is possible to address this problem either by generalizing over an abstract explanation or by using some inductive generalization over the results of EBRL.

In the future, we plan to study how our work generalizes to multiple levels of hierarchy. So far, we have been able to retain optimality while using a hierarchy to speed up convergence of RL. In general, we may have to give up optimality for speed of convergence. It would be nice to be able to automatically make such choices, and choose between different hierarchies or refine a crude hierarchy into a more accurate one.

Acknowledgements

The authors gratefully acknowledge the support of the Office of Naval Research under grant number N00014-95-1-0557 and the support of the National Science Foundation under grant number IRI-9520243.

References

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA.

Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, *5*.

Dietterich, T. G., & Flann, N. (1995). Explanation-based learning and reinforcement learning: A unifed view. In *Proceedings of Machine Learning Conference*.

Kaelbling, L. P. (1993). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of Machine Learning Conference*, pp. 167–173.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Aritificial Intelligence Research*, *4*, 237–285.

Koenig, S., & Simmons, R. G. (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, *22*, 227–250.

Ok, D., & Tadepalli, P. (1996). Auto-exploratory average reward reinforcement learning. In *Proceedings of AAAI-96*.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning,* San Mateo, CA. Morgan Kaufmann.

Shavlik, J. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*, 39–70.

Singh, S. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.