Explanation-Based Learning and Reinforcement Learning: A Unified View

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331
tgd@cs.orst.edu

Nicholas S. Flann
Department of Computer Science
Utah State University
Logan, UT 84322-4205
flann@nick.cs.usu.edu

Abstract

In speedup-learning problems, where full descriptions of operators are always known, both explanation-based learning (EBL) and reinforcement learning (RL) can be applied. This paper shows that both methods involve fundamentally the same process of propagating information backward from the goal toward the starting state. RL performs this propagation on a state-by-state basis, while EBL computes the weakest preconditions of operators, and hence, performs this propagation on a region-by-region basis. Based on the observation that RL is a form of asynchronous dynamic programming, this paper shows how to develop a dynamic programming version of EBL, which we call Explanation-Based Reinforcement Learning (EBRL). The paper compares batch and online versions of EBRL to batch and online versions of RL and to standard EBL. The results show that EBRL combines the strengths of EBL (fast learning and the ability to scale to large state spaces) with the strengths of RL (learning of optimal policies). Results are shown in chess endgames and in synthetic maze tasks.

1 Introduction

Speedup learning is a form of learning in which an inefficient problem solver is transformed into an efficient one. Consider a problem solver that conducts state-space search: beginning at a start state, it tries to find a sequence of operators that will reach a goal state. One way to speed up state-space search is to learn a good policy for choosing operators such that a shortest (or cheapest) path is followed from the start state to the goal. A policy π is a function that maps states to operators—it tells which operator $Op = \pi(s)$ to apply in each state s.

Explanation-based learning (EBL) learns such policies by analyzing successful (or failed) operator sequences. Given a sequence of operators $S = \langle Op_1, Op_2, \ldots, Op_n\rangle$ that reaches the goal G, it computes a set P of similar problems such that the *same sequence of operators* S would reach the goal as well. It can then formulate a control rule which says "If the current state is in P, then apply Op_1." (Similar rules can be learned for each suffix of the operator sequence: $\langle Op_i, \ldots, Op_n\rangle$.) Taken together, these control rules define the learned policy π. Typically, the set P is described intentionally as a *region* in state space—not as a list of individual states. The process of computing the set P—called the *preimage* of the goal G with respect to the operator sequence S—is called *goal regression*, because a description of the goal is "regressed" through models of each of the operators in S.

Consider for example, the LEX2 system (Mitchell, Keller, & Kedar-Cabelli, 1986), which applies EBL to speed up symbolic integration. A state in LEX2 is an expression, such as $\int 5x^2 dx$. The goal is to transform this expression to one that does not contain the integral sign. Two of the available operators are Op_1 : $\int kf(x)dx = k\int f(x)dx$ and Op_2 : If $n \neq -1, \int x^n dx = \frac{x^{n+1}}{n+1}$. The operator sequence $S = \langle Op_1, Op_2\rangle$ solves this problem. Now, working backwards, LEX2 can infer that any state in the set $P = \{\mathtt{e} | \mathtt{match}(\mathtt{e}, \int kx^n dx) \wedge n \neq -1\}$ can be solved by this same operator sequence. The meta-predicate `match(e1,e2)` is true if expression `e1` matches expression `e2` through unification of variables.

The effectiveness of EBL is determined by the cost of performing goal regression and the cost of matching the learned rules. The Prodigy (Minton, 1988) and SOAR (Laird, Rosenbloom, & Newell, 1986) architectures incorporate many features designed to minimize these costs.

Reinforcement learning (RL) is another approach to learning policies for state space search. RL algorithms assume that the state space includes a reward function, R. When operator Op is applied to move from state s to state s', the problem solver receives reward $R(s, Op, s')$. A typical reward function would provide positive reward when s' is a goal state, and a small negative reward (or cost) for each step that does not reach a goal state. An optimal policy is one that maximizes the reward received by the problem solver.

Most reinforcement learning algorithms work by learning an optimal value function and then computing the optimal policy from the value function. The optimal value function $f(s)$ for state s gives the total reward that will be received by following the optimal policy starting in state s. Once f is learned, the optimal policy can be computed by performing a one-step lookahead search (applying each of the possible operators Op_i), evaluating the resulting states, and choosing the operator that leads to the state of highest value.

To learn the value function, RL algorithms begin with an initial value function $\hat{f}$ and incrementally improve it through a series of so-called "Bellman backups" (after Bellman (1957)). A Bellman backup updates the estimated value of a state $\hat{f}(s)$ by performing a one-step lookahead search and backing up the maximum resulting value:

$$\hat{f}(s) := \max_{Op}[R(s, Op, Op(s)) + \hat{f}(Op(s))].$$

It can be shown that regardless of the initial value function $\hat{f}$, if Bellman backups are performed in every state infinitely often, then eventually, $\hat{f}$ will converge to the optimal value function f.

Bellman backups can be performed in any order. Most RL algorithms interleave problem solving with learning. At each state s during problem solving, the algorithm must decide whether to apply the current best operator (as indicated by one-step lookahead using the current value function) or to make an "exploratory" move (i.e., apply some other operator to "see what happens"). In either case, the algorithm can also perform a Bellman backup on state s. Some RL algorithms perform a series of trials. In each trial, a search is conducted to find a sequence S of operators that solves a problem. Then Bellman backups are performed by working backwards from the goal along S. Finally, some RL algorithms work by full offline search. A complete backward search of the state space is conducted starting from the goal state. All operators are applied *in reverse* and Bellman backups are performed at each state. This can be efficiently organized as a best-first search with a priority queue storing states that have not yet been expanded by the reverse search. The priority of each node is its current estimated value according to $\hat{f}$. In each iteration, the algorithm pops the unexpanded state s' of highest value and applies all operators in reverse. For each such operator Op, it checks whether the backed-up value $R(s, Op, s') + \hat{f}(s')$ is greater than the previous value of $\hat{f}(s)$, where $s = Op^{-1}(s')$. If it is, then the value of $\hat{f}(s)$ is updated, and s is pushed onto the priority queue (unless it is already there). If not, then there is no need to push s onto the queue. The initial value of all non-goal states should be set to $-\infty$ for this to work properly. We will call this basic algorithm POINT-DP, since it performs point-based dynamic programming. (This is essentially the same as the Dijkstra algorithm for shortest paths in a graph. See Cormen, Leiserson & Rivest, 1990, p. 527; Dijkstra, 1959).

1.1 Relating EBL and RL

In the terminology of dynamic programming, the control rules learned in EBL represent a policy π that maps from states to operators. Notice, however, that EBL does not compute any value function. As a consequence, EBL is not able to learn optimal policies.

To illustrate this problem, consider again the rule learned by LEX2:

> If the current state matches $\int kx^n dx$
> and $n \neq -1$
> Then apply Op_1.

This policy is not optimal for the state $\int 0x^1 dx$, because there are cheaper operators that can apply. However, once a control rule has been learned, most EBL systems apply that control rule to all future states where it applies.[1] This means that these systems are very sensitive to the quality of the initial operator sequence constructed to solve a new problem. A poor operator sequence will lead to a poor policy.

Because EBL systems do not learn optimal policies, they do not have any need to perform exploratory actions. Even if such actions were to discover a better path to the goal, EBL systems would have difficulty detecting or exploiting this path.

EBL systems do possess an important advantage over RL systems—they can reason with *regions* rather than with *points*. The central problem with standard, state-based reinforcement learning algorithms is that they do not scale to large state spaces. The value function $f(s)$ is typically represented by a large table with one entry for every possible state. The time required for batch dynamic programming is proportional to the number of states, so a large state space places severe time and space limitations on the applicability of dynamic programming and reinforcement learning.

Many researchers have recently investigated methods for introducing some form of "state generalization" that would allow reinforcement learning algorithms to

[1] This is an oversimplification of some EBL systems, but the central point applies: no existing EBL system can guarantee that it learns optimal policies. Instead, various ad hoc methods have been developed to ameliorate this problem.

learn the policy for many states based on experience with only a few states. Perhaps the most popular approach is to represent the value function by some function approximation method, such as local weighted regression (Atkeson, 1990) or a feed-forward neural network (Tesauro, 1992; Sutton, 1988; Lin, 1992). A closely related line of research attempts to partition the state space into regions having the same (or similar) values for the value function (Chapman & Kaelbling, 1991; Moore, 1993). A difficulty with all of these approaches is that they rely on first gathering experience (through problem solving) and then detecting that several states have similar values.

Explanation-based learning provides an alternative approach to state generalization. The goal regression step of EBL is very closely related to the Bellman backup step of RL. A Bellman backup propagates information about the value of a state backwards through an operator to infer the value of another state. Goal regression propagates information about the value of a *set* of states backwards through an operator to infer the value of another *set of states*.

Unlike inductive approaches to state generalization, EBL chooses regions based on the states where a specific operator (or a sequence of operators) is applicable. As with applications of EBL in concept learning, this provides a form of *justified generalization* over states. EBL does not need to gather experience over a region and then make an inductive leap—it can commit to the region by analyzing the sequence of operators applied in a single experience. This ability to reason with regions has permitted EBL to be applied to problems with infinite state spaces, such as traditional AI planning and scheduling domains, where state-based reinforcement learning would be inapplicable (Minton, 1988).

These observations concerning the relationship between EBL and RL suggest that it would be interesting to investigate a hybrid algorithm that could perform *region-based Bellman backups*. These backups would combine the region-based reasoning of EBL with the value function approach of RL. Such a hybrid would also need to adopt some form of exploration in order to find optimal policies. We call this hybrid algorithm *Explanation-Based Reinforcement Learning* (or EBRL).

The remainder of this paper describes online and batch EBRL algorithms and compares them to standard online EBL and to online and batch RL algorithms. We show that EBRL outperforms all of the other algorithms in both batch and online settings. To quantify and predict the performance improvements, we define a parameter, ρ, to be the mean number of states contained in an EBRL region. We show that the performance improvements can be predicted directly from the value of ρ. Finally, we show how EBRL can be applied to the reverse-enumeration of chess endgames to give deeper and more useful endgame tables than batch RL can provide.

2 Methods

We begin by describing a simple robot maze domain that we will employ to illustrate and evaluate our algorithms. Next, we describe five algorithms whose performance we will compare. Finally, we discuss criteria for evaluating the performance of reinforcement learning algorithms.

2.1 Test Domain

Consider the simple maze problem shown in Figure 1. There are six goal states (marked by G's), and any state can be a starting state. The task is to construct an optimal policy for moving from any state to the goal states. There are 16 available operators that can be divided into three groups: *single-step operators* (north, south, east, and west), which take a single step in one of the four directions, *to-wall operators* (north-to-wall, south-to-wall, east-to-wall, west-to-wall), which move as far as possible in one of the four directions until they encounter a wall, at which point they stop, and *wall-following operators* (north-follow-east-wall, north-follow-west-wall, south-follow-east-wall, etc.), which move along a wall until the wall ends. There are eight wall following operators, because an operator must specify which direction it moves the robot and which wall it is following (e.g., "go north following the east wall"). These operators have the following costs: single-step, 1 unit; to-wall, 3 units; and wall-following, 5 units. The robot receives a reward of 100 units when it reaches the goal.

It is important to note that the to-wall and wall-following operators have the "funnel" property—that is, they map many initial states into a single resulting state. In this simple problem, on the average, each operator maps 5.11 states into a single resulting state. The effectiveness of EBL and EBRL is significantly enhanced by funnel operators, because even when the resulting state is a single state, goal regression through a funnel operator yields a *set* of initial states. Without the funnel property, the only way EBL (and EBRL) could reason with regions would be if the goal region contains many states.

Figure 2 shows an optimal policy for this maze problem. A simple arrow in a cell indicates a single-step operator; an arrow that is terminated by a perpendicular line indicates a to-wall operator; and an arrow with a small segment perpendicular to its center is a wall-following operator (and the small segment indicates which wall to follow). The figure shows that there are large regions of state space where the optimal policy recommends the same operator, so we might hope that EBL and EBRL can find those regions easily.

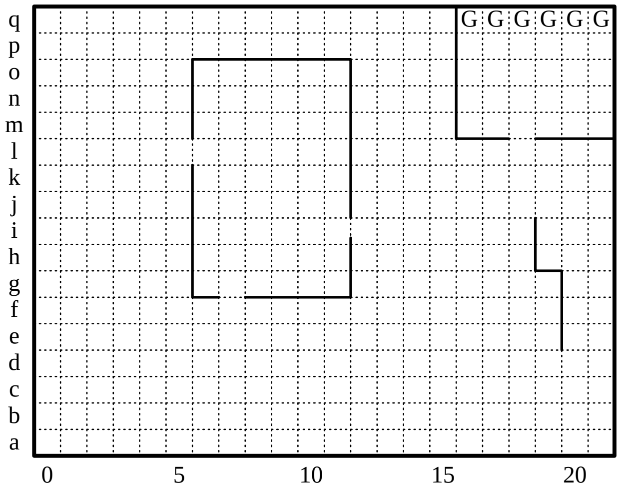

Figure 1: A simple maze problem. "G" indicates a goal state.

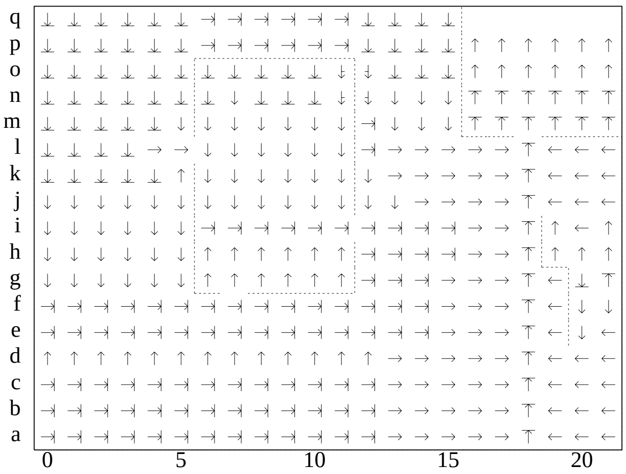

Figure 2: An optimal policy for the maze task. See text for explanation of symbols.

2.2 Algorithms

We have implemented the following five algorithms for speedup learning.

The first algorithm is point-based offline dynamic programming, POINT-DP, which we described in the introduction. Figure 3 shows the value function after one iteration of POINT-DP. The goal state at q16 has been popped off the priority queue and expanded. The value 99 of p16 reflects applying the north operator. The value of 97 in o16, n16, and m16 all reflect applying the north-to-wall operator. Note that other operators, such as west, west-to-wall, and west-follow-north-wall, were all considered, but the backed-up values where smaller than existing values for states such as q17.

The second algorithm is rectangle-based offline dynamic programming (RECT-DP). This is nearly identical to the point-based algorithm except that the value function is represented by a collection of rectangles. The priority queue stores pairs of the form (rectangle, value). Each inverse operator Op^{-1} is able to take a rectangle r' and compute its *preimage* as a disjoint *set* P of rectangles such that applying Op to any state in any of those rectangles will result in a state in rectangle r. The need for each inverse operator to return a *set* of rectangles can be seen in Figure 4, which shows the value function (and the rectangles) after one iteration of the algorithm. Notice that when the goal rectangle is given to the inverse operator north-to-wall^{-1}, three rectangles result: One rectangle from lower left corner m16 to upper right corner q17, one rectangle from a18 to q18, and one rectangle from m19 to q21. Computing these rectangles efficiently requires careful choice of algorithms, but it can be accomplished in time $O(m+\log W)$, where m is the number of resulting rectangles and W is the number of walls in the maze.

In general, the value function f ends up looking like a display of overlapping rectangles on a workstation screen. We can imagine the observer looking down on the maze world. Rectangles with higher f values occlude rectangles with lower f values (ties broken arbitrarily). During the learning process, new rectangles will be added to this data structure.

Algorithms for this problem have been studied in computational geometry. Bern (1990) shows a data structure that requires $O(\log^3 n + c\log^2 n)$ time to insert a new rectangle (where n is the number of rectangles and c is the number of "visible" line segments cut by the new rectangle). This data structure can also retrieve the top-most rectangle at a given point in $O(\log^3 n)$ time. As a side-effect, the algorithm can produce a set of disjoint rectangles to describe the "visible" region of any rectangle. In our case, the number of rectangles will be n/ρ, where n is the total number of states in the state space. The number of visible line segments cut cannot exceed 2ρ. So the time to insert a new rectangle in our algorithms grows as $O(\log^3(n/\rho)+\rho\log^2(n/\rho))$.

The third algorithm we consider is simple asynchronous DP with experience replay (we will call it RL). RL conducts an infinite sequence of trials. In each trial, problem solving begins in a randomly-chosen state. Unless it chooses to make an exploratory move, the problem solver computes the one-step lookahead greedy policy using the current value function. That is, it tries all applicable operators, computes the value of the state resulting from each, and picks the operator that takes it to the state with the highest value. Ties are broken randomly. The sequence of states and operators is stored in a LIFO queue. Once the goal is reached, the sequence is processed in reverse order performing Bellman backups to update the value of each state visited along the path. We implemented a simple counter-based exploration policy for all three of our online algorithms: On the t-th visit to a given state, the greedy policy is followed with probability $t/16$; otherwise a random action is chosen.

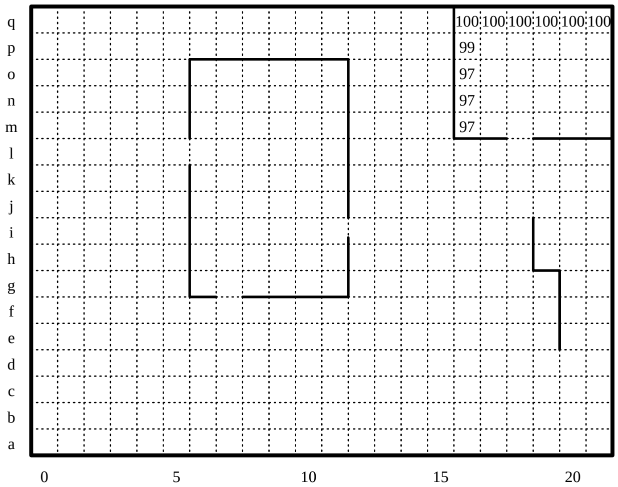

Figure 3: The value function for POINT-DP after one iteration.

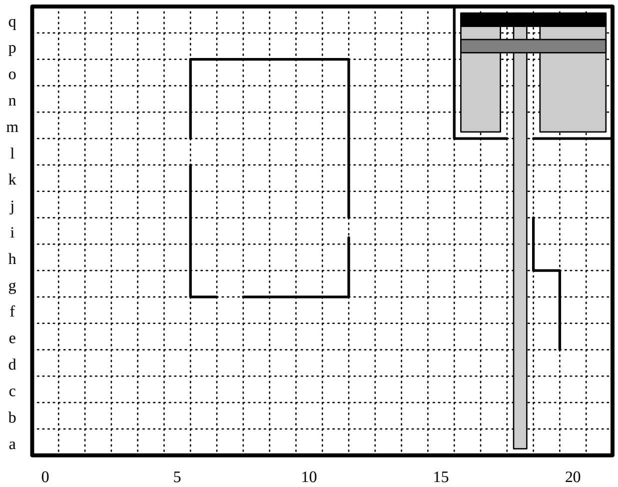

Figure 4: Five boxes created by the first iteration of RECT-DP. Darkest box is the goal box (value 100). Medium grey box has value 99, and the light grey boxes have value 97.

The fourth algorithm we consider is EBL. The EBL procedure performs a series of trials. In each trial, problem solving begins in a randomly-chosen state. Operators are applied to move from this starting state to the goal. The sequence of operators and resulting states is pushed onto a LIFO queue (this is slightly different than RL, where preceding-state/operator pairs were pushed onto the queue). Once the goal is reached, rectangle-based backups are performed by popping (state,operator) pairs off the queue.

Unlike all of the other algorithms in this paper, EBL does not construct a value function f. Instead, it constructs a policy π. The policy is represented as a collection of rectangles. Conceptually, we can think of this collection as a FIFO queue. New rectangles are inserted at the tail of the queue. To find the rectangle that covers a given state s, we start searching from the head of the queue. Hence, we will retrieve the rectangle covering s that was inserted into the queue *earliest*. During problem solving, EBL performs no exploration. Instead, it obeys the current policy π.

The EBL approach of applying the first learned policy rule for state s in all future visits to state s means that the quality of the learned policy is sensitive to the quality of the operator sequence chosen during the first visit to state s. Another way of saying this is that the quality of the learned policy in EBL is determined by the quality of the initial policy. In all of the other four algorithms, the initial policy is entirely random. To give EBL a better initial policy, we extended our algorithm so that it makes ten forward searches (with exploration) to the goal and then performs backups along the best of these ten paths.

The final algorithm we consider is explanation-based reinforcement learning (EBRL). It is essentially the same as the EBL algorithm, except that it learns a value function and a policy rather than just a policy alone. The key change is to replace the π data structure with the f data structure that we presented in the RECT-DP algorithm. New rectangles are inserted into f according to their value (they are "above" all rectangles with a lower value and "below" all rectangles with a higher value).

This completes our description of the five algorithms. We turn now to a discussion of the criteria for evaluating and comparing these algorithms.

2.3 Evaluation Criteria for Speedup Learning

Speedup learning involves a tradeoff between the cost of learning and the quality of performance. If we perform no learning, then performance is usually poor. If we spend a large amount of CPU time in learning, then performance can eventually become optimal. Given this tradeoff, the principal way that we will assess speedup learning algorithms is by plotting the quality of their performance as a function of the amount of learning that has been performed.

We will measure quality in two ways. The first measure is *expected reward*. To measure performance quality, we will take the average undiscounted reward obtained by the problem solver when it is started in every possible start state (and follows the current policy). The second measure is *policy coverage*. The coverage of a policy is the percentage of states for which the current value function has a non-default value. In the maze problems, for example, the value of each state is initialized to -1. The optimal value function has values in the range [88,100]. The coverage of the policy is the number of states for which the value is not equal to -1. Once a state has a non-default value, this means that we have learned *something* about what action to choose in that state. This is usually better than applying the default initial policy to that state.

We will measure learning time in two ways as well. The first measure is the *number of Bellman backups.* This applies to both point-based and region-based algorithms and to both batch and online algorithms. However, while a point-based Bellman backup requires essentially constant time, region-based backups may require longer times (depending on how the regions are represented). The second measure is the *number of training trials.* Online algorithms are trained through a series of trials. A trial consists of choosing one of the start states at random and asking the problem solver to get from the chosen start state to a goal state. Because of the prototype nature of our implementations, we do not measure CPU time, because this could be substantially improved through more painstaking coding. Instead, we will apply results from computational geometry to establish the cost of each Bellman backup and each operator application in a trial.

3 Experiments and Results

3.1 Batch Algorithms

We begin by comparing the two batch algorithms, POINT-DP and RECT-DP on our simple maze problem. Table 1 summarizes the performance of these two algorithms. A backup is "useful" if it results in an improved value for some state.

Table 1: Performance of batch algorithms.

	Bellman Backups	
Algorithm	Total	Useful
POINT-DP	3,156	513
RECT-DP	452	88

In the final f structure for RECT-DP, 88 rectangles were stored, but only 71 of them had any visible states. The value of ρ, the average number of states in each rectangle, was 5.27.

Figure 5 compares the coverage of the two algorithms as a function of the number of useful Bellman backups. We can see that RECT-DP attains coverage much faster than POINT-DP, as we would expect. This is very important in domains (such as chess), where full execution of either algorithm is impossible, because the state space is too large. In such cases, before resources are exhausted, RECT-DP can achieve much higher coverage than POINT-DP.

To understand the scaling behavior of these algorithms, we performed the following two experiments. First, we constructed two random mazes of size 50 by 50 and 100 by 100 with approximately the same value for ρ as our example maze. We then measured the number of Bellman backups. One would expect that the number of backups would be proportional to the number of states n for POINT-DP and proportional to the number of abstract states n/ρ for BATCH-DP. Table 2 and Figure 6 show that this is indeed the case. RECT-DP maintains an advantage of roughly a factor of 7 in the number of backups it performs (and a factor of 5.6—roughly ρ—in the number of backups that result in visible rectangles).

Table 2: Bellman backups as a function of the number of states in the problem.

	POINT-DP Backups		RECT-DP Backups	
States	Total	Useful	Total	Useful
327	3156	513	452	88
2500	20932	3352	2989	608
10000	83968	13998	12525	2494

Table 3: Comparison of performance on the 170-row, 220-column version of the original maze.

Algorithm	Total Backups	Useful Backups
POINT-DP	300840	39725
RECT-DP	1367	277

The second experiment we conducted was to convert our example problem from a 17-row, 22-column maze into a 170-row, 220-column maze by subdividing each original state into 100 new states. This gave a value for ρ of 145.53. Table 3 shows that the cost of running RECT-DP increased by only a factor of 3, while the cost of running POINT-DP increased by a factor of 77 (for useful backups) and 95 (for total backups). For the new problem, RECT-DP computes 220 times fewer backups (and about 143 times fewer visible backups).

Here is a summary of our observations: (a) RECT-DP makes roughly 5 times as many Bellman Backups as it makes useful Bellman Backups in this domain, (b) POINT-DP makes roughly 6 times as many Bellman Backups as it makes useful Bellman Backups in this domain, (c) the ratio of RECT-DP-useful-backups to POINT-DP-useful-backups is very close to ρ, and (d) the ratio of RECT-DP total backups to POINT-DP total backups was roughly 1.3 to 1.5 times ρ From these, we can see that the ρ parameter is critical for predicting the relative number of backups made by RECT-DP and POINT-DP.

The other key factor affecting performance is the cost of performing each backup. We estimate that for POINT-DP, the cost of processing one state scales as $O(B_p \log n)$, where B_p is the branching factor of the search (in the reverse direction), and n is the number of states in the space. For our example maze problem, the POINT-DP branching factor is 8.44 (minimum 4, maximum 80). For RECT-DP, there are added costs for dealing with rectangles. We estimate the cost to scale as $O(B_r(\log^3 n/\rho + \rho \log^2 n/\rho))$. For our example maze problem, the mean branching factor for RECT-DP is 6.37 (minimum 4, maximum 26).

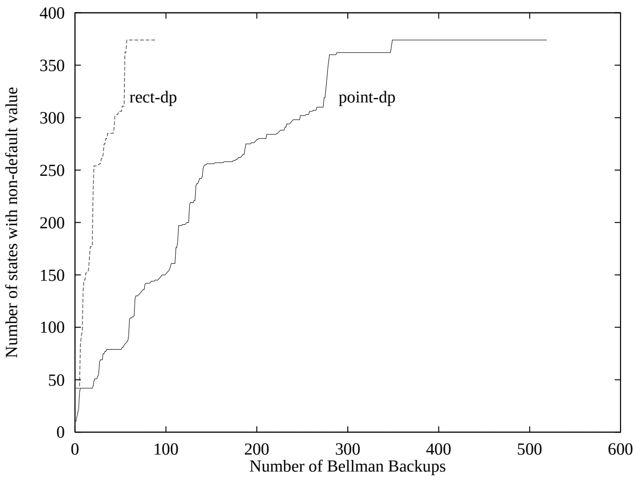

Figure 5: Coverage of batch algorithms as a function of the number of useful Bellman backups.

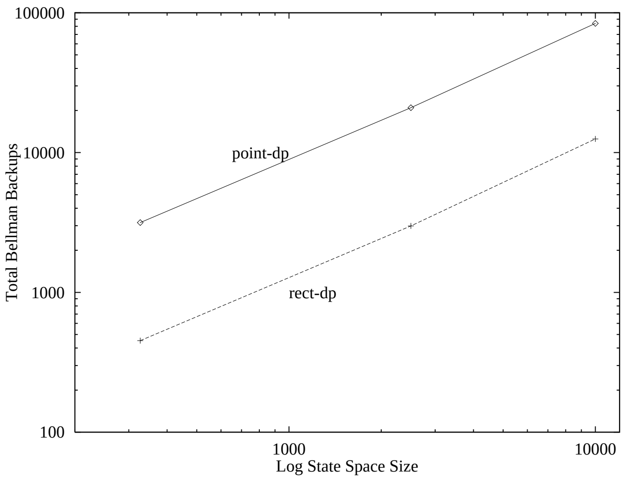

Figure 6: Total number of Bellman backups as a function of the number of states. The number of walls was chosen to keep ρ approximately constant ($\rho_{374} = 5.27; \rho_{2500} = 5.24; \rho_{10000} = 5.20$)

3.2 Online Algorithms

To compare the three online algorithms (RL, EBL, and EBRL), we performed 30 runs of each. Each run consisted of a series of trials. Each trial consisted of a forward search (with exploration) from a random starting state to the goal followed by a series of Bellman backups along the solution path. After each trial, we measured the average undiscounted reward that would be received by starting the problem solver in every possible start state and following the current policy (see Figure 7).

We can see that EBRL reaches optimal performance much faster than either EBL or RL. Furthermore, we can see that EBL converges to a poor performance level. This is a consequence of the fact that EBL can never replace a bad learned rule with a better one (i.e., it can never replace a rectangle of low value with a rectangle of higher value, because it does not learn a value function). The wide error bars on EBL reflect its sensitivity to the solution paths found in the first few trials. Some runs—with good early trials—perform much better than other runs—with very bad early trials.

3.3 RECT-DP and POINT-DP in chess endgames

To demonstrate the generality of the approach, we applied the batch methods to develop optimal policies for playing chess endgames. Chess is considerably more complex than the synthetic maze task, because we have two actors with opposing goals. One player is trying to maximize its return, the other trying to minimize. Both batch algorithms require major modifications to handle this. The most important of these modifications is that Bellman backups for the minimizing player backwards from a state s must be delayed until all backups *into* state s for the maximizing player have been completed. Hence, the algorithms attach to each state (or region) a count of the number of maximizing backups *into* that state remaining to be performed. Backups *from* that state are delayed until the count is zero. Further details are given in Dietterich & Flann (1994).

We studied POINT-DP and RECT-DP applied to the king versus king and rook (KRK) ending in chess. While this ending is one of the simplest, it is difficult to play well, even for experts, and it can involve up to 42-ply of forced moves to win. The value function is derived for the maximizing side (black), the player with the king and the rook. The losing side is white, the player with only the king. The initial goal states describe black-to-play, win-in-one-ply positions where the white king is immediately captured. There are 18,704 possible goal states for POINT-DP and three goal regions for RECT-DP.

The results of running RECT-DP and POINT-DP for the entire KRK chess endgame are shown in Figure 8. Useful Bellman backups were determined differently for the minimizing and maximizing players. For the maximizing player, a useful backup was one that led to a change in value for a state or region. In other words, when a shorter path to a win was identified. For the minimizing player, all backups that reduced the number of remaining backward operators into any state were considered useful.

From Figure 8, it is clear that POINT-DP requires approximately 50 times more backups than RECT-DP (706,584 versus 13,619). One factor that can account for this difference is the representation of the goal states. For POINT-DP, 18,704 states must be initially backed up to identify the 54 lost-in-2-ply positions. This accounts for the initial flat period of the graph where 114,251 backups are required to achieve

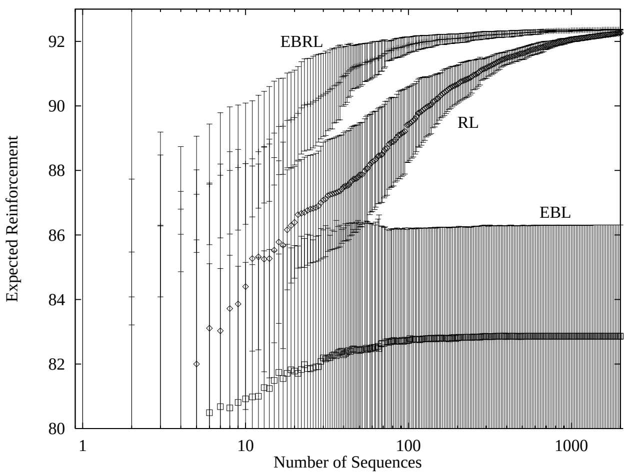

Figure 7: Comparison of expected reward for explanation-based learning (EBL), reinforcement learning (RL), and explanation-based reinforcement learning (EBRL). Error bars show one standard error on each side of the mean of 30 runs.

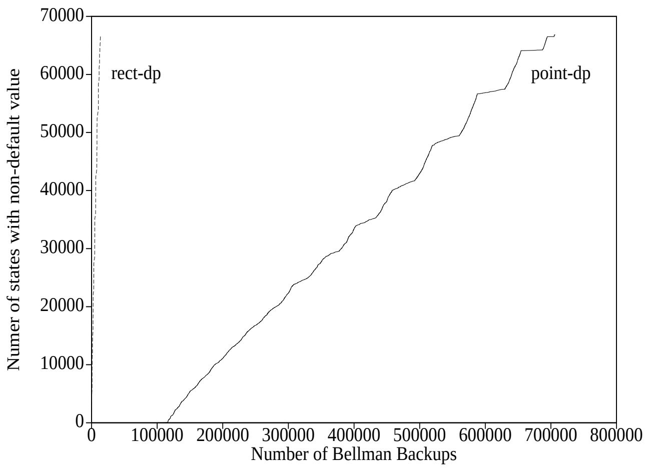

Figure 8: Coverage of batch algorithms as a function of the number of useful Bellman Backups for KRK chess endgame.

a coverage of 54. RECT-DP, on the other hand, must only backup three general regions, which enables rapid progress to be made, with only 254 backups needed to reach the same coverage. A further reason why the region-based abstraction is so effective in this domain is because of the "funnel" property of the rook moves.

Similar results have been generated for other, more complicated endings in chess and in checkers (see Flann, 1992, for more details).

4 Discussion

The results clearly show that explanation-based reinforcement learning (RECT-DP and EBRL) out-performs point-based dynamic programming (POINT-DP and RL) and explanation-based learning (EBL) in 2-D maze tasks. The region-based methods achieve better coverage and better mean reward in less time than the point-based methods. The results also show that RECT-DP out-performs POINT-DP in reverse-enumeration of chess endgames. RECT-DP achieves better coverage before exhausting available resources.

Two key factors determine the relative performance of region-based methods over point-based methods. The first factor is ρ—the average number of points in each region produced by RECT-DP. If the point-based state space contains n states, then the region-based state space contains n/ρ states. Experimentally, we observed that the number of useful Bellman backups performed by the region-based methods is reduced from the number performed by point-based methods by a factor of ρ.

The second key factor is the cost of reasoning with regions. For rectangles in two dimensions, this cost is roughly $O(\log^3(n/\rho) + \rho \log^2(n/\rho))$. Edelsbrunner (1983) describes a rectangle-tree data structure that provides reasonably efficient reasoning with d-dimensional rectangles. However, in discrete-valued spaces with many dimensions, the costs may be much higher (Tambe, Newell, & Rosenbloom, 1990). These high costs are the primary cause of the "Utility Problem" of explanation-based learning (Minton, 1990; Subramanian & Feldman, 1990). Some researchers have explored algorithms that combine region-based policies with a default policy (Minton, 1988). This has the advantage of reducing the number of rectangles that need to be stored and manipulated (but at the cost of eliminating the ability to learn an optimal policy).

Region-based methods are limited by the need for the learning system to have complete models of the effects of each operator. If operator models are incorrect, then the resulting policy will not be optimal. For example, if operators are missing preconditions, then preimages computed during Bellman backups will be too large. If operators are missing effects, then the post-images will be too large, so backups will be performed on inappropriate states.

Applying region-based methods where the operators are stochastic is an idea that has been explored in robotics (e.g., Lozano-Perez, Mason, & Taylor, 1984; Erdmann, 1986). Because we have been interested primarily in problem solving domains where the operators are deterministic computational steps, we have not studied stochastic operators. We speculate that exact region-based methods with stochastic operators will not be computationally attractive, but that combinations of region-based backups with function approximation methods could be promising.

5 Conclusion

We have shown that the fundamental step in Explanation-Based Learning—computing the preimage of a region with respect to an operator—is closely related to the Bellman backup step of dynamic programming and reinforcement learning algorithms. Based on this insight, we designed and implemented batch and online algorithms that perform region-based Bellman backups. We compared these to standard batch and online algorithms that perform point-based backups and showed that the region-based methods give a substantial improvement in performance.

This analysis moves us one step closer to building a unified theory of speedup learning for problem-solving systems. It also helps us understand some of the problems that have been observed in EBL systems such as Soar and Prodigy. In particular, the region-based algorithms show how to design EBL systems that can learn optimal policies for problem solving.

The central challenge facing reinforcement learning is to find ways of scaling up to large state spaces. Region-based dynamic programming provides one possible solution to this problem, and perhaps it will suggest other possible solutions in the near future.

Acknowledgements

The authors gratefully acknowledge the partial support of NSF grants IRI-9204129 and IRI-8657316, NASA Ames Research Center grant NAG 2-630, and ONR contract number N00014-95-1-0557. The authors also thank Prasad Tadepalli, David Cohn, Eric Chown, and Steve Minton for useful discussions.

References

Atkeson, C. G. (1990). Using local models to control movement. In *Advances in Neural Information Processing Systems*, Vol. 2, pp. 316–323. Morgan Kaufmann.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press.

Bern, M. (1990). Hidden surface removal for rectangles. *J. Comp. and Sys. Sci.*, *40*, 49–69.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proc. 12th IJCAI*, pp. 726–731. Morgan Kaufmann.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to Algorithms.* MIT Press.

Dietterich, T. G., & Flann, N. S. (1994). Explanation-based learning and reinforcement learning: A unified view. Tech. rep., Oregon State University, Corvallis, OR.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Edelsbrunner, H. (1983). A new approach to rectangle intersections. *Int. J. Comp. Math.*, *13*, 209–219.

Erdmann, M. (1986). Using backprojections for fine motion planning with uncertainty. *Int. J. Rob. Res.*, *5*(1), 19–45.

Flann, N. S. (1992). *Correct Abstraction in Counter-planning: A Knowledge Compilation Approach.* Ph.D. thesis, Oregon State University.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in Soar: the anatomy of a general learning mechanism. *Mach. Learn.*, *1*(1), 11–46.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, *8*, 293–322.

Lozano-Perez, T., Mason, M. M., & Taylor, R. H. (1984). Automatic synthesis of fine-motion strategies for robots. *Int. J. Rob. Res.*, *3*.

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach.* Ph.D. thesis, Carnegie-Mellon University. Tech. Rep. CMU-CS-88-133.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Art. Int.*, *42*, 363–392.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Mach. Learn.*, *1*(1), 47–80.

Moore, A. W. (1993). The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In *Advances in Neural Information Processing*, Vol. 6, pp. 711–718 San Mateo, CA. Morgan Kaufmann.

Subramanian, D., & Feldman, R. (1990). The utility of ebl in recursive domain theories. In *Proc. AAAI-90* Menlo Park, CA. AAAI Press.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Mach. Learn.*, *3*(1), 9–44.

Tambe, M., Newell, A., & Rosenbloom, P. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Mach. Learn.*, *5*, 299–348.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Mach. Learn.*, *8*, 257–278.