Machine Learning, 28, 169–214 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Explanation-Based Learning and Reinforcement Learning: A Unified View

THOMAS G. DIETTERICH tgd@cs.orst.edu
Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202

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

Editor: Andrew Barto

Abstract. In speedup-learning problems, where full descriptions of operators are known, both explanation-based learning (EBL) and reinforcement learning (RL) methods 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. Most RL methods perform this propagation on a state-by-state basis, while EBL methods compute the weakest preconditions of operators, and hence, perform this propagation on a region-by-region basis. Barto, Bradtke, and Singh (1995) have observed that many algorithms for reinforcement learning can be viewed as asynchronous dynamic programming. Based on this observation, this paper shows how to develop dynamic programming versions of EBL, which we call region-based dynamic programming or Explanation-Based Reinforcement Learning (EBRL). The paper compares batch and online versions of EBRL to batch and online versions of point-based dynamic programming and to standard EBL. The results show that region-based dynamic programming combines the strengths of EBL (fast learning and the ability to scale to large state spaces) with the strengths of reinforcement learning algorithms (learning of optimal policies). Results are shown in chess endgames and in synthetic maze tasks.

Keywords: Explanation-based learning, reinforcement learning, dynamic programming, goal regression, speedup learning, incomplete theory problem, intractable theory problem

1. Introduction

Speedup learning is a form of learning in which an inefficient problem solver is transformed into an efficient one. It is often easy to specify and implement an inefficient problem solver for a task, whereas implementing an efficient problem solver can be very difficult. For example, in the game of chess, an inefficient problem solver can be implemented as an exhaustive search algorithm that applies the rules of the game. An efficient problem solver would need to transform those rules into a near-optimal policy for choosing moves in the game. Similarly, the problem of job-shop scheduling can be solved by a simple problem solver that generates and tests all possible schedules. An efficient problem solver must exploit particular properties of the job shop and the job mix to find efficient search heuristics. There are many important applications that could benefit from effective speedup learning algorithms.

1.1. Explanation-Based Learning

In the field of machine learning, the best-studied speedup learning method is Explanation-Based Learning (EBL), as exemplified by the Prodigy (Minton, 1988) and SOAR (Laird, Rosenbloom, & Newell, 1986) systems. EBL systems model problem solving as a process of state-space search. The problem solver begins in a start state, and by applying operators to the start state and succeeding states, the problem solver seeks to reach a goal state, where the problem is solved. The problem solver in EBL systems is typically initialized with one of the weak methods, such as means-ends analysis or heuristic search. It then applies its weak methods to solve problems. The key step of EBL is to analyze a sequence of operators S that solved a problem and compute the set P of similar problems such that the *same sequence of operators* S would solve those problems as well. This set P of similar problems is then captured as a control rule which states

If the current state is in P
Then apply the sequence of operators S.

This analytical process is sometimes called "goal regression," because the goal is regressed through the sequence of operators to compute P.

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

$$Op_1: \quad \int kf(x)dx = k\int f(x)dx, \text{ and}$$

$$Op_2: \quad \text{If } n \neq -1, \int x^n dx = \frac{x^{n+1}}{n+1}.$$

The operator sequence $S = (Op_1, Op_2)$ solves this problem. Now, working backwards, LEX2 can infer that any state in the set $P = \{\int kx^n dx \wedge n \neq -1\}$ can be solved by this same operator sequence.

Note that the process of computing the set P requires complete and correct models of the effects of each of the operators available to the problem solver.[1] Note also that the computation of the set P can be performed very efficiently for some kinds of operator representations as long as the set P can be represented intensionally.

A variation on EBL that is employed in LEX2 and SOAR is to learn a control rule for each state along the sequence of operators $S = (Op_1, Op_2, \ldots, Op_n)$ that solved a particular problem. The result is a list of sets, $(P_1, P_2, \ldots, P_n)$. Each set P_i describes those states such that the sequence of operators $(Op_i, Op_{i+1}, \ldots, Op_n)$ will reach a goal state. The following collection of control rules is then created (one for each value of i):

If the current state is in P_i
Then apply operator Op_i.

This kind of control rule—which maps from a set of states to a single operator—will be the focus of our attention in this paper.

1.2. Reinforcement Learning

Substantial progress has recently been made in another area of machine learning: Reinforcement Learning (RL). Like EBL, reinforcement learning systems also engage in state-space search. However, unlike EBL, RL systems typically do not have models of their operators. Instead, they learn about their operators by applying them and observing their effects. Another difference between RL and EBL is that RL systems seek to maximize a "reward" rather than to reach a specified goal state. Each time an RL system applies an operator Op to move from a state s to a resulting state s', it receives some real-valued reward, $R(s, Op, s')$. These rewards are typically summed to define the cumulative reward (or *return*) received by the RL system.[2] The goal of RL is to learn a policy for choosing which operator to apply in each state so that the return of the system is maximized. Formally, a *policy* is a function (denoted π) that maps from states to operators. Hence, when a problem solver is pursuing a particular policy π in state s, it applies the operator $\pi(s)$.

Despite these differences between EBL and RL, RL methods can also be applied to solve speedup learning problems. Given a speedup learning problem, we can define a reward function as follows. For each operator, Op, we can provide a reward equal to the negative of the cost (in CPU time) of applying Op. When the problem solver reaches a goal state, we can provide a fixed reward (e.g., zero) and terminate the search (i.e., the goal states are absorbing states). With this reward function, the cumulative reward of a policy is equal to the negative of the cost of solving the problem using that policy. Hence, the optimal policy will be the policy that solves the problem most efficiently. Reinforcement learning problems of this form are called stochastic shortest-path problems.

In the remainder of this paper, we will focus on the application of RL methods to speedup learning problems under this kind of reward function. For the most part, we will be concerned with deterministic operators, since most speedup learning problems involve only such operators. We will define the return of a policy to be the cumulative reward. To study the generality of our methods, however, we will also explore problems with stochastic operators, in which case the return of a policy will be the expected cumulative reward. We will assume that there exists a non-zero probability path from every starting state to a goal state.

Given any policy π, it is useful to compute a second function, called the *value function* f^π, of the policy. The value function tells, for each state s, what return will be received by applying the policy π in s and then continuing to follow π indefinitely. Many RL algorithms work by learning the value function rather than directly learning a policy. This approach is possible because there are dynamic programming algorithms (discussed below) for taking f^π and incrementally modifying it to produce a value function $f^{\pi'}$ corresponding to a better policy. By apply-

ing these improvements repeatedly, the value function will converge to the optimal value function (denoted f^*).

Once the optimal value function has been computed, the optimal policy (denoted π^*) can be computed by a one-step lookahead search as follows. Let $Op(s)$ be the state that results (in the deterministic case) from applying operator Op to state s. Then, $\pi^*(s)$ can be defined as

$$\pi^*(s) = \operatorname*{argmax}_{Op} \left[R(s, Op, Op(s)) + f^*(Op(s))\right].$$

The expression $R(s, Op, Op(s))$ is the reward for applying operator Op in state s, and $f^*(Op(s))$ is the return of pursuing the optimal policy starting in state $Op(s)$. In other words, the optimal policy applies the operator that results in the highest return according to the optimal value function f^* (plus the reward for applying the operator itself).

An important advantage of RL over EBL is that RL algorithms can be applied to domains with stochastic operators. Specifically, let $p(s'|Op, s)$ be the probability that if we apply operator Op in state s we will move to state s'. Then the value function f^π gives the *expected value* of the policy π:

$$f^\pi(s) = \sum_{s'} p(s'|\pi(s), s)[R(s, \pi(s), s') + f^\pi(s')]$$

Barto, Bradtke & Singh (1995) have shown that many RL algorithms can be analyzed as a form of asynchronous dynamic programming. The fundamental step in most dynamic programming algorithms is called the "Bellman backup" (after Bellman, 1957). A Bellman backup improves the estimated value of a state $f(s)$ by performing a one-step lookahead search and backing up the maximum resulting value:

$$f(s) := \max_{Op} \sum_{s'} p(s'|Op, s)[R(s, Op, s') + f(s')] \tag{1}$$

It can be shown (for stochastic shortest-path problems) that regardless of the initial value function f, if Bellman backups are performed in every state infinitely often, then eventually, f will converge to the optimal value function f^*. Based on this result, a simple table-based dynamic programming algorithm can work as follows: (a) represent the value function f as a large table with one cell for every state; (b) initialize f to zero; (c) repeatedly choose a state s at random and perform a Bellman update of s to compute a new value for $f(s)$; (d) repeat until convergence.

In the case where the operators are deterministic, the Bellman backup has the following simpler form:

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

Different RL algorithms perform their Bellman backups in different orders. The standard RL algorithms are online algorithms that interleave problem solving with

learning. At each state s during problem solving, an online 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. This takes advantage of the fact that after computing the current best operator (through one-step lookahead search), no further computation is needed to do the Bellman backup. Exploratory moves are essential. Without them, it is easy to construct situations where the optimal policy will not be found.

A slight variation on this online approach is to perform "reverse trajectory" backups. With reverse trajectory backups, no Bellman backups are performed during problem solving (but the sequence of operators is stored). Once the goal state is reached, the Bellman backups are performed in reverse order, starting with the last operator applied and working backwards through the operator sequence. This can give more rapid learning, because the f values being backed up are more up-to-date than in the standard online approach. However, this may also introduce inefficiencies, because the value backed-up along the trajectory into state s may no longer be the highest value that could be obtained by a one-step lookahead search from s. This can be avoided by repeating the one-step lookahead search at each state s before performing the backup. Lin (1992) employed a somewhat more complex version of reverse trajectory updates in a simulated robot problem.

A third way that Bellman backups can be applied is by an offline search technique known as prioritized sweeping (Moore & Atkeson, 1993). The central idea is that whenever we update the value of a state s', we apply all of the available operators *in reverse* to generate a list of states (the predecessors of s') whose values might need to be updated to reflect the (updated) value of s'. We push this list of "update candidates" onto a priority queue, sorted by the magnitude of the potential change to their values. At each iteration, we pop off the state s whose value has the potential for greatest change, and we perform a Bellman backup on that state. (This requires performing a one-step lookahead search and computing the backed-up value, as shown in Equation (1).) If this results in a change in $f(s)$, we compute the predecessors of s, and push them onto the priority queue. We initialize the priority queue with the predecessors of each of the goal states.

If the operators are deterministic, each of these three methods can be substantially simplified. The key is to initialize the value of every state to be $-\infty$ and maintain the invariant that the estimated value of every state is always less than or equal to its true value. If this is invariant holds, then for any state s and operator Op, we can perform a partial Bellman backup without considering any of the other operators that might be applied to s:

$$f(s) := \max\{f(s), R(s, Op, Op(s)) + f(Op(s))\}.$$

In essence, we are incrementally computing the one-step lookahead search and taking the maximum. This was first discovered by Dijkstra (1959), so we will call this

single-operator backup a "Dijkstra backup." If a Dijkstra backup leads to a change in the value of $f(s)$, we will call it a "useful" backup.

Dijkstra backups can be performed either in the forward, online algorithm or in the reverse trajectory algorithm. The deterministic version of prioritized sweeping is Dijkstra's shortest path algorithm (Cormen, Leiserson, & Rivest, 1990). This algorithm takes advantage of the fact that when we compute a predecessor s of a state s', we can compute $R(s, Op, s') + f(s')$, and order the priority queue (in decreasing order) by these backed-up values. This ensures that we will not consider backing up a value from a state s' to one of its predecessors until all successors of s' that could possibly raise the value of s' have been processed. This in turn means that a backup is performed at most once for each state-operator pair. Unfortunately, this property does not carry over to the stochastic case.

This review of RL algorithms has focused on algorithms that employ dynamic programming to learn a value function. There are many other approaches to RL (see Kaelbling, Littman & Moore, 1996, for a review). Some methods explicitly learn a policy (either with or without learning a value function), which permits them to avoid the one-step lookahead search needed to choose actions when only a value function is learned. Another important algorithm is Q-learning (Watkins, 1989; Watkins & Dayan, 1992), which learns a value function $Q(s, a)$ for state-action pairs. The function $Q(s, a)$ gives the expected value of performing action a in state s.

1.3. Relating EBL and RL

In the terminology of dynamic programming, the control rules learned in EBL represent a (partial) 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, whereas RL methods are able to learn optimal policies (at least in principle).

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.[3] 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 table-based RL systems—they can reason with *regions* rather than with *points*. The central problem with

point-based RL 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 imposes severe time and space limitations on the applicability of dynamic programming and RL.

Many researchers have investigated methods for introducing some form of "state generalization" or "state aggregation" that would allow RL algorithms to 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; Bertsekas & Castanon, 1989; Sammut & Cribb, 1990). A difficulty with all of these approaches is that they rely on first gathering experience (through problem solving), inferring values for some of the states, and then discovering regularities in those values. An added difficulty is that in most cases, the optimal value function cannot be exactly represented by the function approximation method. This can prevent the algorithms from finding the optimal policy.

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 point-based RL would be inapplicable (Minton, 1988).

These observations concerning the relationship between EBL and RL suggest that it would be interesting to investigate hybrid algorithms that could perform *region-based backups*. These backups would combine the region-based reasoning of EBL with the value function approach of RL. The resulting set of regions would provide an *exact* representation of the value function, rather than an approximate representation based on some state aggregation scheme. We call these hybrid algorithms *Explanation-Based Reinforcement Learning* (or EBRL) algorithms.

Some researchers have previously explored region-based backups in RL tasks. Yee, Saxena, Utgoff, and Barto (1990) described a system that performs online RL using a kind of region-based backup. They organized the regions into trees of "concepts"

with exceptions, and developed methods that attempt to find large, useful regions. Their system out-performed a non-learning problem solver that conducted 6-ply lookahead search in tic-tac-toe.

The remainder of this paper describes online and batch EBRL algorithms and compares them to standard online EBL and to online and batch RL (dynamic programming) algorithms. We show that the EBRL algorithms outperform all of the other algorithms in both batch and online settings in both deterministic and stochastic problems. 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 the algorithms we are comparing: five algorithms for deterministic problems and two algorithms for stochastic problems. Finally, we discuss criteria for evaluating the performance of RL 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 a goal state. There are 16 available operators that can be divided into three groups:

Single-step operators: north, south, east, and west. These four operators take one step in one of the four directions.

To-wall operators: north-to-wall, south-to-wall, east-to-wall, west-to-wall. These operators move as far as possible in one of the four directions until they encounter a wall, at which point they stop.

Wall-following operators: north-follow-east-wall, north-follow-west-wall, and so on. These operators are only applicable next to a wall. The robot moves along the 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 different costs. The single-step operators cost 1 unit; the to-wall operators cost 3 units; and the wall-following operators cost 5 units. The robot receives a reward of 100 units when it reaches the goal. The goal is to find the policy that maximizes the total reward received by the robot.

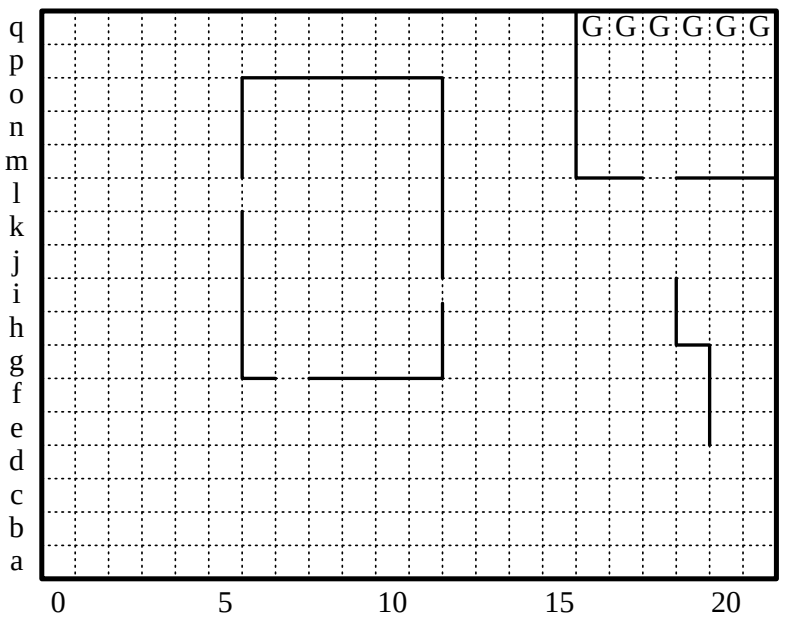

Figure 1. A Simple Maze Problem. "G" indicates a goal state.

It is important to note that the to-wall and wall-following operators have what Christiansen (1992) calls 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 is being followed). 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.

Although this maze problem is very simple and involves maximizing only total reward to a fixed goal state, the region-based (EBRL) methods described in this paper should also work in the discounted reward and average reward cases. The key idea of EBRL is to analyze the (known) models of the operators and the reward function to perform region-based backups. This idea applies to any RL problem,

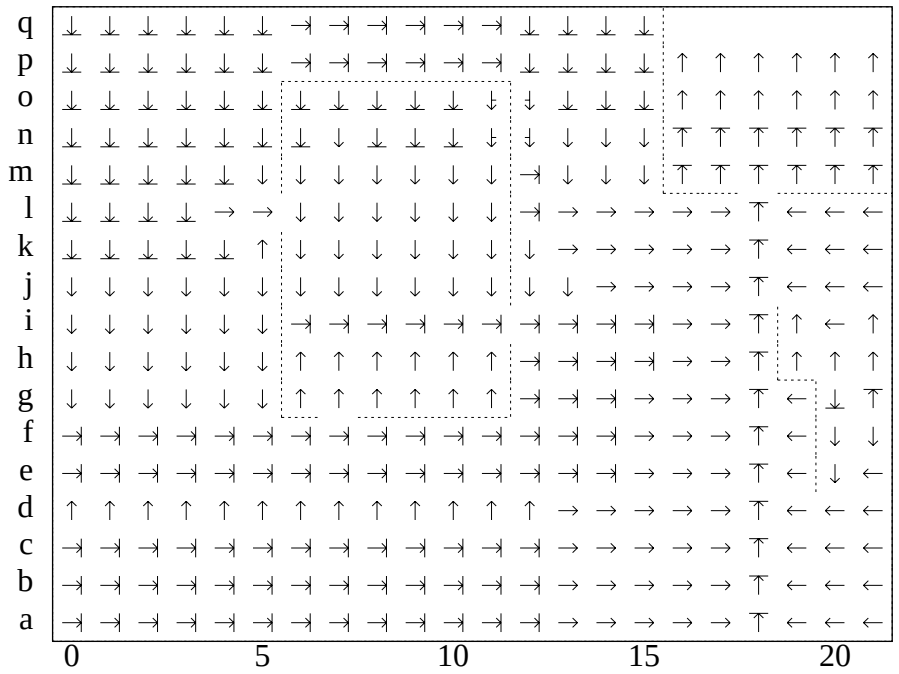

Figure 2. An optimal policy for the maze task. See text for explanation of symbols.

although the effectiveness of region-based backups depends crucially on the nature of the operators and the reward function. Because each region-based backup is equivalent to performing a set of point-based backups, we conjecture that EBRL methods will converge to the optimal policy under the same conditions as point-based dynamic programming methods.

2.2. Algorithms

We first describe five algorithms for deterministic problems. We then describe two algorithms that we have implemented for stochastic problems.

2.2.1. Point-Based Offline Dynamic Programming (OFFLINE-POINT-DP)

Table 1 describes the point-based dynamic programming algorithm, OFFLINE-POINT-DP. It conducts a uniform cost search working backward from the goal. There are two central data structures: An array f representing the value function,

```
1   Let f be the value function (represented as an array
2         with one element for each possible state initialized to −∞).
3   Let Q be a priority queue.
4   Let G be the set of goal states.
5   Let v_g be the reward received when reaching goal state g.
6   For each g ∈ G, push (g, v_g) onto the priority queue Q.
7   While notEmpty(Q) do
8         Let (s', v') := pop(Q).
9         For each operator Op do begin
10              Let P := Op^{-1}(s') be the set of states such that
11                    applying Op results in state s'.
12              For each s ∈ P do begin
13                    Let v := v' − cost(Op) be the tentative backed-up value of state s.
14                    If v > f[s] then begin
15                          f[s] := v
16                          push (s, v) onto Q.
17                          end // if
18                    end // s
19              end // Op
20        end // while
```

Table 1. The OFFLINE-POINT-DP algorithm for offline, point-based dynamic programming.

and a priority queue, Q. This version of offline DP requires an inverse operator Op^{-1} for each operator Op. $Op^{-1}(s')$ returns the set of all states s such that $Op(s) = s'$.

The Dijkstra backup step is performed in lines 12–15, where we compute the backed-up value of state s, determine whether it is better than the best previous value for s, and update $f[s]$ if so. Note that the same state may be pushed onto the priority queue more than once. Hence, the algorithm can be slightly optimized to include a test after line 8 to determine whether the value of state s', $f[s']$, is still equal to v'. If not, then a better backed-up value for s' has been determined (and already popped off the priority queue!), so state s' does not need to be further processed.

Figure 3 shows the value function after one iteration of the While loop at line 7. The goal state at cell *q16* has been popped off the queue and expanded. The value 99 of cell *p16* reflects applying the north operator. The value of 97 in cells *o16*, *n16*, and *m16* all reflect applying the north-to-wall operator. Note that during the loop in lines 9–19, 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*.

OFFLINE-POINT-DP converges to the optimal policy in time proportional to the number of states times the number of operators.

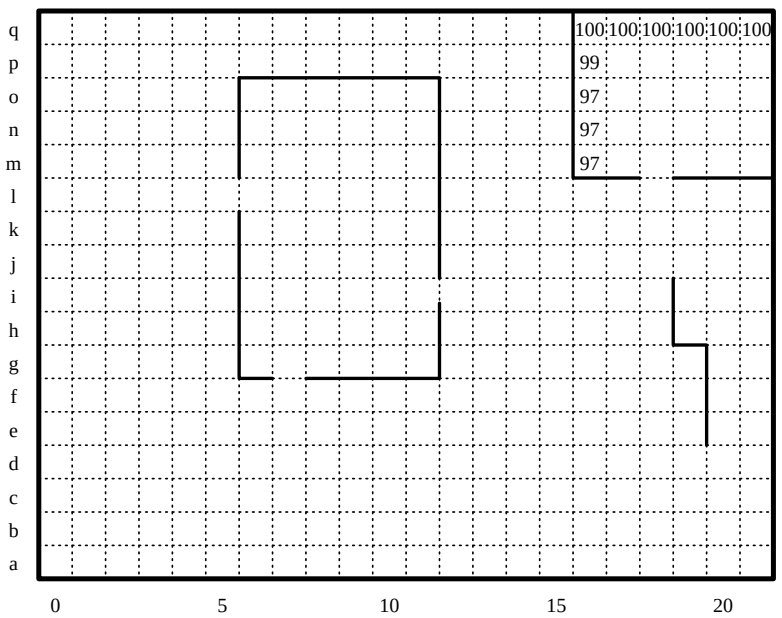

Figure 3. The value function for OFFLINE-POINT-DP after one iteration.

2.2.2. Rectangle-based offline dynamic programming (OFFLINE-RECT-DP)

We now turn to the first of our explanation-based reinforcement learning algorithms, OFFLINE-RECT-DP. Table 2 shows the algorithm for rectangle-based batch dynamic programming. 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 the example shown in Figure 4, which shows the value function after one iteration of the While loop. 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. Horizontal walls can be stored in a kind of 2-d tree so that range queries can be answered in time logarithmic in the number of horizontal walls and linear in the number of relevant walls. The range query is constructed by defining a rectangle that extends from *a16* to *q21*. By proper organization of the 2-d tree, the answers

```
1  Let f be the value function (represented as a collection
2        of rectangles; each rectangle has an associated value and operator.)
3  Let Q be a priority queue.
4  Let G be a set of rectangles describing the goal states.
5  Let v_G be the reward received when reaching a goal state.
6  For each g ∈ G do begin
7        push (g, v_G) onto the priority queue Q.
8        insert (g, v_G, nil) into the collection representing f
9        end // g
10 While notEmpty(Q) do
11       Let (r', v') := pop(Q).
12       For each operator Op do begin
13             Let P := Op^{-1}(r') be a disjoint set of rectangles such that
14                   applying Op to any state in those rectangles results in
15                   a state in rectangle r'.
16             For each rectangle r ∈ P do begin
17                   Let v := v' − cost(Op) be the tentative backed-up value of rectangle r.
18                   If rectangle r with value v would increase the value
19                   of f for any state then begin
20                         insert (r, v, Op) into the collection representing f
21                         push (r, v) onto Q.
22                         end // if
23                   end // r
24             end // Op
25       end // while
```

Table 2. The OFFLINE-RECT-DP algorithm for offline, rectangle-based dynamic programming.

to the query can be retrieved in top-to-bottom order. The first wall that intersects the query rectangle is the wall separating *l16-17* from *m16-17*, and it is used to construct the first rectangle. The query rectangle then shrinks to *a18-q21*. The second nearest wall is the wall separating *l19-21* from *m19-21*, and it is used to construct the third rectangle. The query rectangle then shrinks to *a18-q18*. This rectangle does not intersect any more walls, so it defines the second rectangle. An analogous data structure is required for vertical walls. The cost to evaluate this kind of iterated range query is $O(m + \log W)$, where m is the number of resulting rectangles and W is the number of walls in the maze.

One additional factor complicates the computation of operator preimages. Consider the rectangle *a18-q18*, and suppose we want to compute the preimage of this rectangle with respect to the operator east-to-wall. This operator can only result in states *h18* and *i18*, because none of the other states in rectangle *a18-q18* has a wall on its east side. Hence, to perform the preimage computation, we must first find the subrectangles of *a18-p18* that have an east wall. There is only one such rectangle in this case: rectangle *h18-i18*. We will call such rectangles, "postimage rectangles," because they are in the postimage of the operator in question. All postimage rectangles can be obtained by a range query into the vertical wall data

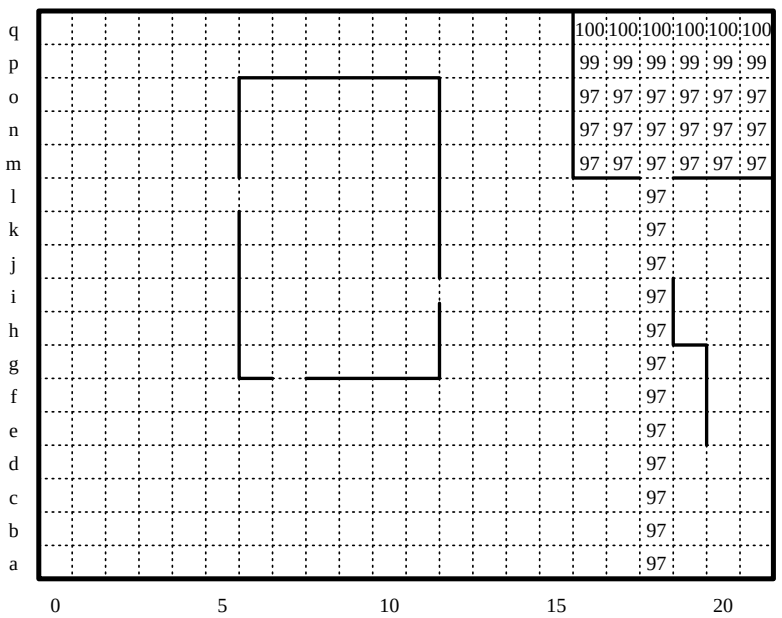

Figure 4. Value function after one iteration of OFFLINE-RECT-DP.

structure mentioned above. Once we have these rectangles, we can compute their preimages with respect to east-to-wall. In this example, this produces two new rectangles: *i6-i16* and *h12-h16*.

Figure 5 shows the rectangles that have been constructed by the first Dijkstra backup of OFFLINE-RECT-DP. Notice that the three light-grey rectangles produced by the preimage of north-to-wall are hidden "beneath" the medium-gray rectangle (*p16-p21*) produced by the preimage of the north operator (and also beneath the dark-gray goal rectangle, *q16-q21*). 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

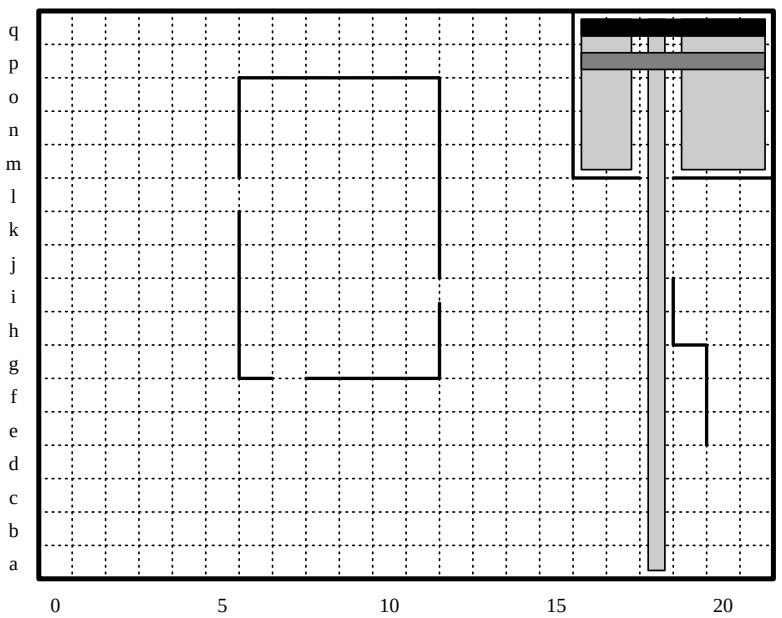

Figure 5. Five boxes created by the first iteration of OFFLINE-RECT-DP. Darkest box is the goal box (value 100). Medium grey box has value 99; the goal can be reached by the north operator from these states. The light grey boxes have the value 97; the goal can be reached by the north-to-wall operator from any of these states.

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 fact that rectangles overlap suggests that a potential speedup in the algorithm could be obtained after line 11 by replacing r' by the set of disjoint sub-rectangles of r' that are currently "visible." If no such rectangles exist, then r' can be discarded.

It is important to note that although this algorithm is expressed in terms of rectangles, the same approach is suitable to any problem where regions can be represented intensionally. For example, expressions in propositional logic are equivalent to hyperrectangles in a high-dimensional space with one dimension for each proposition symbol. Expressions in first-order logic provide an even more powerful representation for regions. The computational geometry algorithms referred to above do not apply to these higher-dimensional representations, but the OFFLINE-RECT-DP algorithm works without modification as long as a data structure can be implemented that represents a collection of regions with attached priorities and that can efficiently determine the region of highest priority that contains a given point.

```
1    Let G = the goal states.
2    Let f be the value function (represented as an array).
3    Let path be a LIFO queue.
4    Repeat forever
5         path := nil
6         s := random state.
7         while s ∉ G do
8              choose the operator Op to apply
9              push (s, Op) onto path
10             s := Op(s)
11             end
12        s' := s
13        while path not empty do
14             pop (s, Op) from path
15             let v := f(s') − cost(Op)
16             if v > f(s) then f(s) := v
17             s' := s
18             end
19        end
```

Table 3. Simple Reinforcement Learning with Reverse Trajectory Updates: ONLINE-POINT-DP

2.2.3. Reinforcement Learning (ONLINE-POINT-DP)

We implemented a simple asynchronous dynamic programming algorithm for reinforcement learning (with reverse trajectory updates) as shown in Table 3. This algorithm, which we will call ONLINE-POINT-DP, conducts an infinite sequence of trials. In each trial, problem solving begins in a randomly-chosen state. Unless the problem solver chooses to make an exploratory move, the problem solver computes the one-step lookahead greedy policy using the current value function. That is, it applies 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 after subtracting off the cost of the operator. Ties are broken randomly. The sequence of states and operators is stored in the LIFO queue *path*. Once the goal is reached, the sequence is processed in reverse order performing Dijkstra backups (lines 13–18).

We implemented the following counter-based exploration policy: We maintain a counter for each state that tells how many times we have visited that state. Suppose we have visited a state t times. Then with probability $t/16$, we will follow the greedy policy when we enter that state. Otherwise, we will choose an operator at random from among the operators applicable in that state. The value 16 was chosen to be the number of operators. In practice, performance is quite good well before we have visited every state 16 times.

A feature of all three of our online algorithms is that they can recover from operator sequences that visit the same state multiple times before reaching the

Table 4. Example Operator Sequence to Illustrate Path Optimization

from state	operator	reward (or negative cost)
l19	east	−1
l20	west	−1
l19	south	−1
k19	west	−1
k18	north	−1
l18	north-to-wall	−3
q18	none (goal)	100

goal. Consider, for example, the sequence of operators (starting in state l19) shown in Table 4.

This sequence is suboptimal, because the second operator just undoes the effects of the first operator. One might worry that the final backed-up reward for state *l19* would be 92 (= 100 − 8). However, there is no need to find and remove inefficiencies of this kind from the sequence. They are handled automatically by the Dijkstra backup in lines 15–16. The backups are performed in reverse order, so the backed-up value of *l18* is 97, *j18* is 96, *j19* is 95, *l19* is 94, and *l20* is 93. When we pop the last state-operator pair, (*l19*,east) off the stack, we compute $v = 92$, but then, in line 16, we look up the best known value for state s = l19, and we find that it is 94, so $f(l19)$ is not modified.

2.2.4. Explanation-Based Learning

Table 5 shows our implementation of the EBL algorithm. As with the ONLINE-POINT-DP algorithm, 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). During the learning phase (lines 14–23), 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*. (In a good implementation, π would be implemented by the more efficient data structure used in the OFFLINE-RECT-DP algorithm above.)

During problem solving, EBL performs no exploration. Instead, it obeys the current policy π. Hence, in line 10, the operator to apply is chosen according to π, if π recommends an operator, or at random otherwise.

```
1   Let G = a set of rectangles containing the goal states.
2   Let π be the policy function (represented as as a collection of rectangles;
3         each rectangles has an associated operator.)
4   Let path be a LIFO queue.
5   For each rectangle g ∈ G, insert g into π.
6   Repeat forever begin
7         s := random state.
8         path := nil
9         while s ∉ G do // Forward search to goal.
10              choose the operator Op to apply
11              s := Op(s)
12              push (Op, s) onto path
13              end
14        while path not empty do // Performs backups along path.
15              pop (Op, s') from path
16              Let r' := the first rectangle in π covering s'
17              Let P := Op^-1(r') be a disjoint set of rectangles such that
18                    applying Op to any state in those rectangles results in
19                    a state in rectangle r'.
20              For each r ∈ P do begin
21                    insert (r, Op) into the collection representing π
22                    end // r
23              end // path
24        end
```

Table 5. Explanation-Based Learning

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 modified the code in Table 5 to repeat the forward search in lines 9–13 ten times and use the path with the best reward to carry out the backups in lines 14–23.

Like ONLINE-POINT-DP, our EBL algorithm can optimize the operator sequence during the backup phase. This is somewhat surprising, given that EBL does not construct a value function or perform true Dijkstra backups. In fact, EBL discovers many more optimizations along an operator sequence than ONLINE-POINT-DP does (at least initially). To see how this works, consider again the operator sequence in Table 4. When EBL backs up the last operator, it constructs a rectangle (*a18-q18*) that selects operator north-to-wall. When it pops off the next operator-state pair (north, *l18*), it looks up *l18* (at line 16) and retrieves the entire rectangle *a18-q18*. It then computes the preimage of this rectangle as the rectangle *a18-q17*. This rectangle is then added to the tail of the π rectangle collection. Because the rectangle *a18-q18* was inserted into π first, however, this new rectangle *a18-p18*

will always be invisible, so the policy π in state *k18* will be to move north-to-wall directly instead of taking one step north first.

Continuing with this example, we pop the pair (west, *k18*) off the stack. The rectangle-backup results in two rectangles *a19-g19* and *j19-q19*. Some EBL systems would generate only the second rectangle, since it covers the state *k19* where the operator west was applied. However, our implementation computes full preimages of each operator application, so these two rectangles are added to π.

Next, we pop the pair (south, *k19*). When EBL looks up *k19* in π, it finds that it is covered by the rectangle *j19-q19*, so it computes the preimage of that rectangle and produces two rectangles: *i19-l19* and *n19-q19*. These are inserted into π, but again, these are both invisible, so they do not actually change the policy.

Next, we pop the pair (west, *l19*). When EBL looks up *l19*, it again retrieves the rectangle *j19-q19*. The backup through west results in the rectangle *j20-q20*, which is added to π.

Finally, we pop the pair (east, *l20*). When EBL looks up *l20*, it retrieves the rectangle *j20-q20*, and computes the preimage *j19-q19* to insert into π. However, this rectangle will be completely invisible. The resulting policy π contains only four visible rectangles:

rectangle	operator
a18-q18	north-to-wall
a19-g19	west
j19-q19	west
j20-q20	west

If we applied π to the same starting state, *l19*, it would solve the problem in two steps: west, north-to-wall, which is the optimal action sequence.

The reason EBL is able to do so well is two-fold. First, the rectangle backups mean that before we even get to state *k18* (during the backup process), we already have a policy for that state from a previous backup. This effectively excises any irrelevant inefficient steps from the operator sequence. The second reason is that the EBL method of placing new rectangles at the tail of the π data structure tends to place them properly with respect to their values, even though backed-up values are not computed. This is because each operator application incurs a cost, so rectangles constructed later (in the backup process) produce lower rewards. When we look up the value of π for a given state, we retrieve the rectangle that was constructed earliest, and—at least during a single trial—that rectangle will be the one with the highest expected reward.

Unfortunately, when EBL is applied in subsequent trials, *all* new rectangles will be placed behind the rectangles constructed in earlier trials, even if those new rectangles have better values. So, within a single trial, EBL optimizes properly, but between trials, it does not.

```
1   Let G = a set of rectangles containing the goal states.
2   Let f be the value function (represented as as a collection of rectangles;
        each rectangles has an associated value and operator).
3   Let path be a LIFO queue.
4   For each rectangle g ∈ G, insert g into f.
5   Repeat forever begin
6       s := random state.
7       path := nil
8       while s ∉ G do // Forward search to goal.
9           choose the operator Op to apply
10          s := Op(s)
11          push (Op, s) onto path
12          end
13      while path not empty do // Perform backups along path.
14          pop (Op, s') from path
15          Let r' := the best rectangle in f covering s'
16          Let P := Op^-1(r') be a disjoint set of rectangles such that
                applying Op to any state in those rectangles results in
                a state in rectangle r'.
17          Let v := f(s') − Cost(Op) be the backed-up value of each state in P
18          For each r ∈ P do begin
19              insert (r, v, Op) into the collection representing f
20              end // r
21          end // path
22      end
```

Table 6. Online region-based dynamic programming (ONLINE-RECT-DP)

2.2.5. Online Region-Based Dynamic Learning (ONLINE-RECT-DP)

Table 6 shows pseudo-code for our online EBRL algorithm, ONLINE-RECT-DP. 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 OFFLINE-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 for deterministic problems. Now we consider versions of the two offline algorithms that can be applied to stochastic problems.

2.2.6. Stochastic Point-Based Offline Dynamic Programming (STOCHASTIC-OFFLINE-POINT-DP)

Table 7 describes our implementation of prioritized sweeping for point-based dynamic programming. The main loop pops a state s off of the priority queue Q,

```
1   Let f be the value function (represented as an array
2         with one element for each possible state initialized to −∞).
3   Let Q be a priority queue.
4   Let G be the set of goal states.
5   Let v_G be the reward received when reaching a goal state.
6   For each g ∈ G, push (g, v_G) onto the priority queue Q.
7   While notEmpty(Q) do
8         Let (s, δ) := pop(Q).
9         Let v := max_Op Σ_s' p(s'|Op, s)[f[s'] − cost(Op)] be the backed-up value of state s.
10        Let δ := |v − f[s]| be the magnitude of the change in value.
11        Let f[s] := v.
12        For each operator Op do begin
13              Let P := Op^−1(s) be the set of states such that
14                    applying Op results in state s.
15              For each s'' ∈ P do begin
16                    push (s'', δ · p(s|Op, s'')) onto Q
17                    end // s''
18              end // Op
19        end // while
```

Table 7. The STOCHASTIC-OFFLINE-POINT-DP algorithm for offline, point-based dynamic programming in stochastic problems.

performs a full Bellman backup on that state (line 9), and then generates the predecessors of s, computes how much the values of those predecessors are likely to change, and pushes each of them onto Q.

2.2.7. Stochastic Region-Based Offline Dynamic Programming (STOCHASTIC-OFFLINE-RECT-DP)

Table 8 describes an algorithm for prioritized sweeping for region-based dynamic programming. It is essentially the same as the point-based algorithm except that the process of performing stochastic region-based backups is much more complex in line 9.

Figure 6 illustrates the process of computing a region-based Bellman backup with stochastic operators. The figure shows the effect of applying operator Op to region R. There are two possible resulting regions. With probabiliy 0.2, the result is $R1$. With probability 0.8, the result is $R2$. Let us focus on region $R1$ first. If we intersect this region with the known regions of highest value, we find two regions: one with value 10 and one with value 20. To back up these values, we compute the preimages of these two regions, intersect them with R, and multiply their values by 0.2 (the probability of reaching these regions). The result is that we subdivide R into two preimage regions (labeled 2 and 4 in the figure). The process for region $R2$ is analogous, and it results in subdividing R into two rectangles labeled 6.4 and 9.6. We now compute the cross-product of these regions to produce the four

```
1   Let f be the value function (represented as set of rectangles).
2       It initially contains one rectangle covering the entire space with value −∞.
3   Let Q be a priority queue.
4   Let G be a list of regions describing the goal states.
5   Let v_G be the reward received when reaching a goal state.
6   For each g ∈ G, push (g, v_G) onto the priority queue Q.
7   While notEmpty(Q) do
8       Let (r, δ) := pop(Q).
9       Let R = {⟨r_1, v_1⟩, ⟨r_2, v_2⟩, ..., ⟨r_k, v_k⟩} be a list of region/value pairs such that
            ∀ s ∈ r_k  v_k = max_Op Σ_s' p(s'|Op, s)[f[s'] − cost(Op)]
10      For each ⟨r_i, v_i⟩ ∈ R
11          Let δ := |v_i − f(r_i)| be the magnitude of the change in value.
12          Insert region r_i into f with value v_i.
13          For each operator Op do begin
14              Let P = Op^{-1}(r_i) be a disjoint set of rectangles such that
15                  applying Op to any state in P results in a state in r_i.
16              For each region r ∈ P do begin
17                  push (r, δ · p(r_i|Op, r)) onto Q
18                  end // r
19              end // Op
20          end // ⟨r_i, v_i⟩
21      end // while
```

Table 8. The STOCHASTIC-OFFLINE-RECT-DP algorithm for offline, region-based dynamic programming in stochastic problems.

rectangles shown at the bottom of the figure. The states in the upper left rectangle have expected value 10.4, which is 4 + 6.4, because with probability 0.2, operator Op will result in a region of value 20 and with probability 0.8, it will result in a region of value 8.

Each of these rectangles can be inserted into the data structure representing the value function. If a different operator Op' results in regions of higher value, those regions will "hide" these regions.

Obviously, there is a great potential for the original region R to be shattered into many small regions with distinct estimated values. Notice that the resulting set of rectangles for R is determined by the size and complexity of the rectangles resulting from applying Op to R. This is sensitive to the exact order in which the backups are performed. For example, it could be the case that after performing this backup for region R, a new high-value rectangle is found that would completely cover $R2$. If we waited to perform the backup on R until this time, then R would only be subdivided into two regions (resulting from $R1$).

We implemented this algorithm and found that in general, the rectangles were shattered into very small regions. Hence, we developed the following alternative approach. First, convert the stochastic operators into deterministic ones by taking only the highest-probability outcome from each operator. Next, apply the OFFLINE-RECT-DP algorithm to this deterministic problem to produce a set of

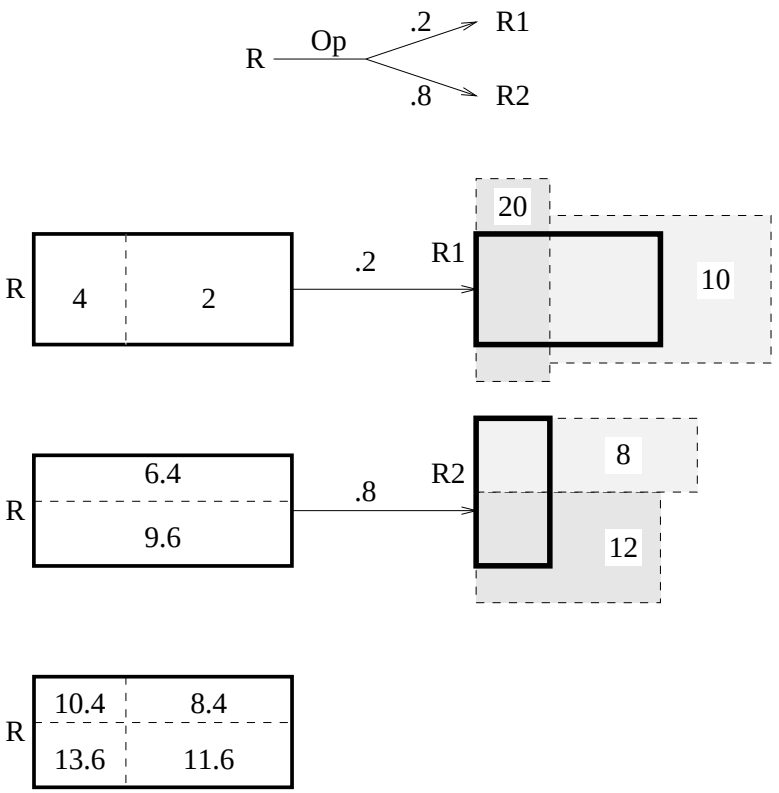

Figure 6. Performing a region-based Bellman backup with stochastic operator *Op*.

rectangles. Then, perform synchronous dynamic programming as shown in Table 9 with the stochastic operators. By doing the updates synchronously, the rectangles being created within one iteration do not cause additional shattering in that same iteration. This approach produced larger rectangles, so the results we report below employed this method.

It should be noted that both of the algorithms (from Figure 8 and Figure 9) converge to the optimal policy. The difference is that the second procedure is able to represent the value function using larger rectangles.

We turn now to a discussion of the criteria for evaluating and comparing these algorithms.

```
1   Let f be the value function (represented as set of rectangles).
2   Repeat forever
3       Let f_new be a new value function (represented as a set of rectangles), initially empty.
3       For each visible rectangle r in the data structure representing f.
4           Let {⟨r_1, v_1⟩, ⟨r_2, v_2⟩, ..., ⟨r_k, v_k⟩} be a list of region/value pairs such that
5               ∀ s ∈ r_k  v_k = max_Op Σ_s' p(s'|Op, s)[f[s'] − cost(Op)]
6           Insert each ⟨r_j, v_j⟩ into f_new.
7           end // for r
8       f := f_new
9       end // repeat forever
```

Table 9. Synchronous region-based dynamic programming.

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 three ways:

1. **Expected cumulative reward**. In each problem, there is a set of possible start states. To measure performance quality, we will take the expected cumulative reward obtained by the problem solver averaged over each of these possible start states.

2. **Deviation from optimal value function**. Another measure of quality is to count the number of states in the state space where the difference between the current estimated value and the value under the optimal policy is greater than some threshold. We will use a threshold of 1, which is the cost of a single move. Several authors have noted that the greedy policy may be optimal even when the value function has not yet converged on the optimal value function. Nonetheless, measurements of the accuracy of the value function are very inexpensive to take, so we will use them as well. To compute a value for states where no value has yet been learned (i.e., no backups have been performed), we will use the mean of the values of all states where a value *has* been learned.

3. **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

Table 10. Performance of batch algorithms.

Algorithm	Dijkstra Backups	Useful Dijkstra Backups
OFFLINE-POINT-DP	3,156	513
OFFLINE-RECT-DP	452	88

in that state. This is usually better than applying the default initial policy to that state.

We will measure learning time in two ways:

1. **Number of backups**. This applies to both point-based and region-based algorithms and to both batch and online algorithms. However, while a point-based backup requires essentially constant time, region-based backups may require longer times (depending on how the regions are represented).

2. **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 will not measure CPU time, because this could be substantially improved through more painstaking coding. Instead, we will apply results from symbolic computing and computational geometry to establish the cost of each backup and each operator application in a trial.

3. Experiments and Results

3.1. Batch Algorithms for Deterministic Problems

We begin by comparing the two batch algorithms, OFFLINE-POINT-DP and OFFLINE-RECT-DP on our simple maze problem. Table 10 summarizes the performance of these two algorithms. The number of Dijkstra backups is equal to the number of times line 13 of OFFLINE-POINT-DP and line 17 of OFFLINE-RECT-DP are executed. The number of useful Dijkstra backups is the number of times lines 15-16 of OFFLINE-POINT-DP and lines 20-21 of OFFLINE-RECT-DP are executed. The backups that were useful resulted in improved values for some state.

In the final f structure for OFFLINE-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 7 compares the coverage of the two algorithms as a function of the number of useful Dijkstra backups. We can see that OFFLINE-RECT-DP attains coverage much faster than OFFLINE-POINT-DP, as we would expect. This is very important in domains (such as chess), where full execution of either algorithm is impossible,

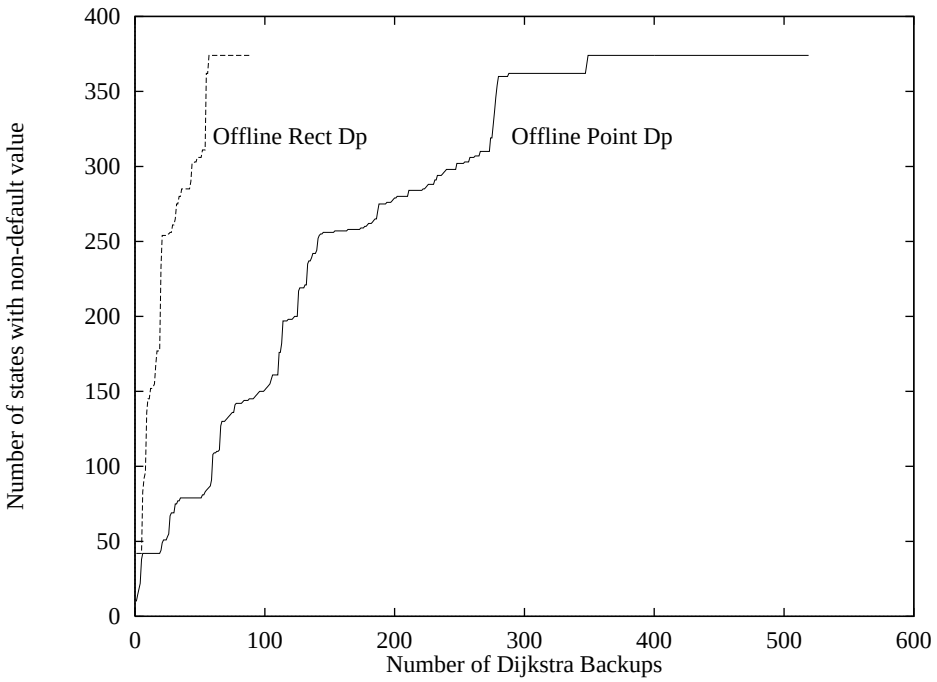

Figure 7. Coverage of batch algorithms as a function of the number of useful Dijkstra backups.

because the state space is too large. In such cases, before resources are exhausted, OFFLINE-RECT-DP can achieve much higher coverage than OFFLINE-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 Dijkstra backups. One would expect that the number of backups would be proportional to the number of states n for OFFLINE-POINT-DP and proportional to the number of abstract states n/ρ for BATCH-DP. Table 11 and Figure 8 show that this is indeed the case. OFFLINE-RECT-DP maintains a constant factor 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).

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 12 shows that the cost of running OFFLINE-RECT-DP increased by only a factor of 3, while the cost of running OFFLINE-POINT-DP increased by a factor of 77 (for useful

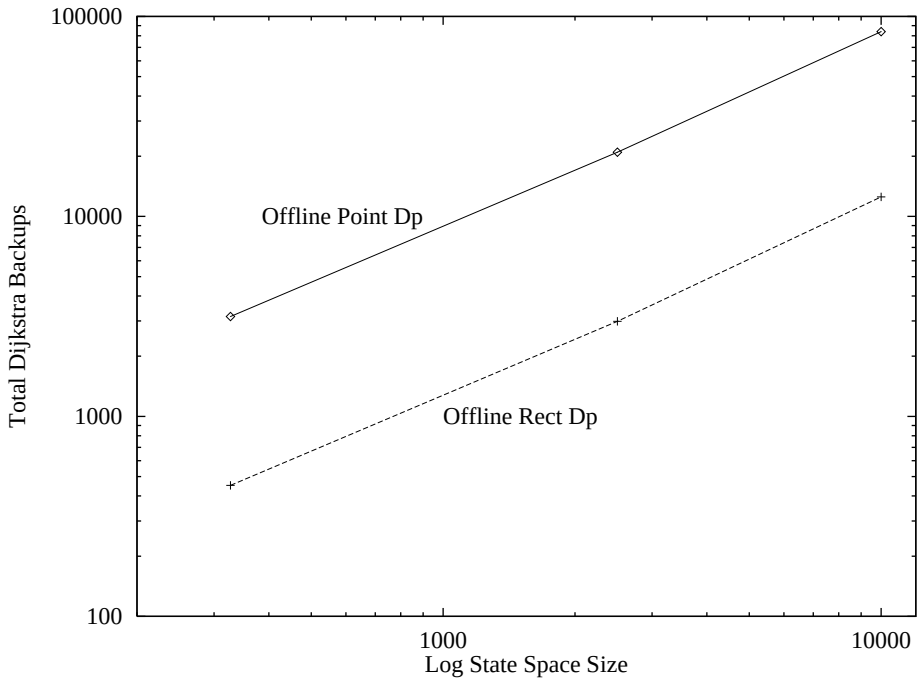

Figure 8. Total number of Dijkstra backups as a function of the number of states. The number of walls was chosen to keep ρ approximately constant ($\rho_{374} = 5.27$; $\rho_{2500} = 5.24$; $\rho_{10000} = 5.20$)

Table 11. Dijkstra backups as a function of the number of states in the problem.

Number of states	OFFLINE-POINT-DP		OFFLINE-RECT-DP	
	Total backups	Useful backups	Total backups	Useful backups
327	3,156	513	452	88
2,500	20,932	3,352	2,989	608
10,000	83,968	13,998	12,525	2,494

Table 12. Comparison of performance on the 170-row, 220-column version of the original maze.

Algorithm	Dijkstra Backups	Useful Dijkstra Backups
OFFLINE-POINT-DP	300,840	39,725
OFFLINE-RECT-DP	1,367	277

backups) and 95 (for total backups). For the new problem, OFFLINE-RECT-DP computes 220 times fewer backups (and about 143 times fewer visible backups).

Here is a summary of our observations:

1. OFFLINE-RECT-DP makes roughly 5 times as many Dijkstra Backups as it makes useful Dijkstra Backups in this domain.

2. OFFLINE-POINT-DP makes roughly 6 times as many Dijkstra Backups as it makes useful Dijkstra Backups in this domain.

3. The ratio of OFFLINE-RECT-DP useful backups to OFFLINE-POINT-DP useful backups is very close to ρ.

4. The ratio of OFFLINE-RECT-DP total backups to OFFLINE-POINT-DP total backups was roughly 1.3 to 1.5 times ρ.

From this, we can see that the ρ parameter is critical for predicting the relative number of backups made by OFFLINE-RECT-DP and OFFLINE-POINT-DP.

The other key factor affecting performance is the cost of performing each backup. For OFFLINE-POINT-DP, the process of performing a backup involves several steps (see Table 1, lines 8–19). The steps involve popping a state-value pair (s', v') off the queue, applying all operators (in reverse) to generate the preimage of that state, considering each state s in the preimage, computing the backed-up value v of s, and (if the value is an improvement), updating the value of s and pushing the state-value pair (s, v) onto the priority queue. Let B_p be the branching factor of OFFLINE-POINT-DP (i.e., the total number of states in the preimages of all operators applied in reverse to a single state s'). The time required for these steps can be summarized as follows:

Step	Cost
pop (s', v') from Q	$O(\log \|Q\|)$
generate all predecessor states	$O(B_p)$
compute their backed-up values	$O(B_p)$
push them onto the Q if necessary	$O(B_p \log \|Q\|)$

Experimentally, $|Q|$ scales as n, the number of states, so we estimate the sum of these costs is $O(B_p \log n)$. For our example maze problem, the OFFLINE-POINT-DP branching factor is 8.44 (minimum 4, maximum 80).

For OFFLINE-RECT-DP, the backup process is basically the same (see Table 2), but there are added costs for dealing with rectangles. As with OFFLINE-POINT-DP, the dominant cost is the cost of inserting new rectangles into the priority queue and inserting them into the f data structure. Let B_r be the branching factor for OFFLINE-RECT-DP (i.e., the total number of preimage rectangles of all operators applied in reverse to a single rectangle r'). Assuming we employ the overlapping rectangle data structure of Bern (1990), these costs are $O(B_r \log n/\rho)$ and $O(B_r(\log^3 n/\rho + \rho \log^2 n/\rho))$ respectively. The latter value dominates, so we estimate the cost of processing each state in OFFLINE-RECT-DP as $O(B_r(\log^3 n/\rho + \rho \log^2 n/\rho))$. For our example maze problem, the mean branching factor for OFFLINE-RECT-DP is 6.37 (minimum 4, maximum 26).

3.2. Batch Algorithms for Stochastic Problems

We now consider what happens when we change the operators in the maze problem so that they are stochastic. We introduced the following form of stochasticity: When an operator is selected for execution, with some probability a different operator is executed instead. Specifically, with probability 0.8, the selected operator is in fact executed, but with probability 0.2, one of the other two operators having a direction perpendicular to the chosen operator is executed instead. For example, if the north operator is chosen, then with probability 0.8, it is executed. With probability 0.1, the east operator is executed instead, and with probability 0.1 the west operator is executed instead. Similarly, if the north-to-wall operator is chosen, then the actual operator executed is north-to-wall (probability 0.8), east-to-wall (probability 0.1), or west-to-wall (probability 0.1). It rarely makes sense to substitute a different wall following operator, so we removed the wall-following operators from the problem.

To evaluate the algorithms, we measured the number of states whose value is more than 1.0 away from the optimal value function. Figure 9 shows the results. We see that the region-based dynamic programming procedure converges to the optimal value function much more quickly than the point-based method as a function of the number of Bellman backups. The final value of ρ in this problem is 1.24 (corresponding to 301 final regions), which shows that in stochastic domains, the effectiveness of region-based methods is much lower than in deterministic domains.

The stochastic substitution of operators moving at right angles to the desired operator creates a large amount of fragmentation of the regions. This results partly from the fact that the domain features (e.g., row and column) important for moving vertically are different from those for moving horizontally. In many applications, it is unlikely that stochastic operators would have this property. Rather, it is likely that errors in the execution of operators would leave most of the same features irrelevant. For example, no matter what errors are introduced when you steer your car, they have no effect on the location of your office or the type of vehicle you are driving.

A way of exploring this within our simple maze problem is to insist that the operators randomly substituted for the selected operator are operators that move in the opposite direction from the selected operator. Hence, when the north operator is selected, the operator actually executed is north (with probability 0.8) or south (with probability 0.2), and so on. Figure 10 shows the results. Region-based dynamic programming has a more substantial advantage here. The final value of ρ is 2.49 (154 regions).

On the other hand, if we modify the operators so that the distance moved is stochastic, which is a form of randomness typically found in robotic applications, the effectiveness of region-based backups is destroyed. This form of randomness destroys the funnel property of the operators, and this causes all of the regions to shrink to single states.

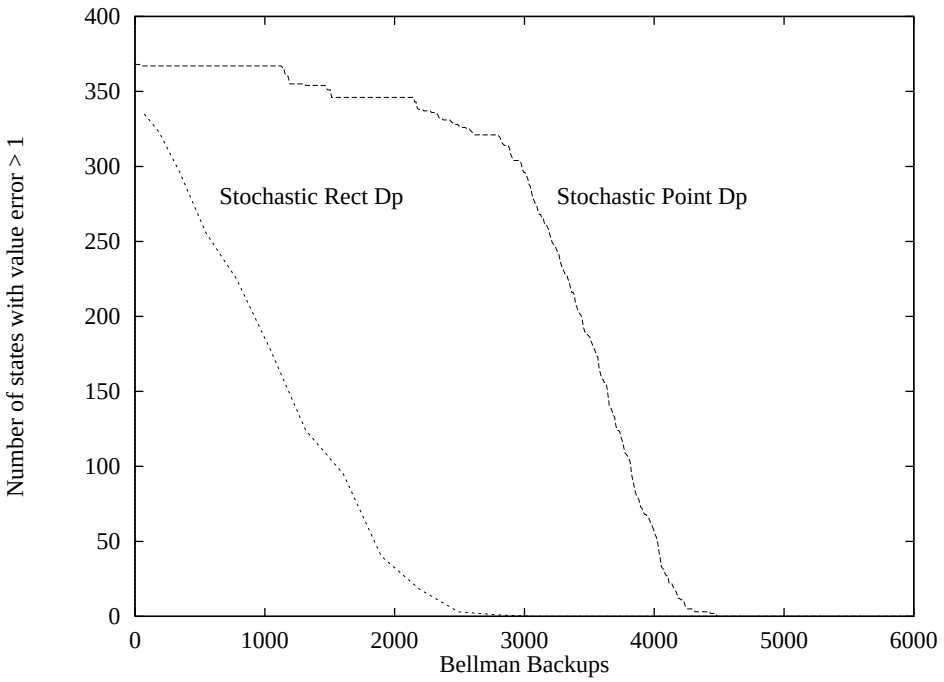

Figure 9. Value function error during dynamic programming—point-based methods versus region-based methods. Randomly-perturbed operators move in a direction perpendicular to the selected operator

From this, we can conclude that the effectiveness of region-based methods depends on the nature of the stochastic behavior of the operators. In some domains, region-based backups will still be worthwhile. However, we have also seen that to make region-based backups work well, we needed to perform backups synchronously. Additional research would be needed to make stochastic region-based backups effective in online settings where synchronous updates are infeasible.

3.3. Online Algorithms

To compare the three online algorithms (ONLINE-POINT-DP, 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 Dijkstra 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

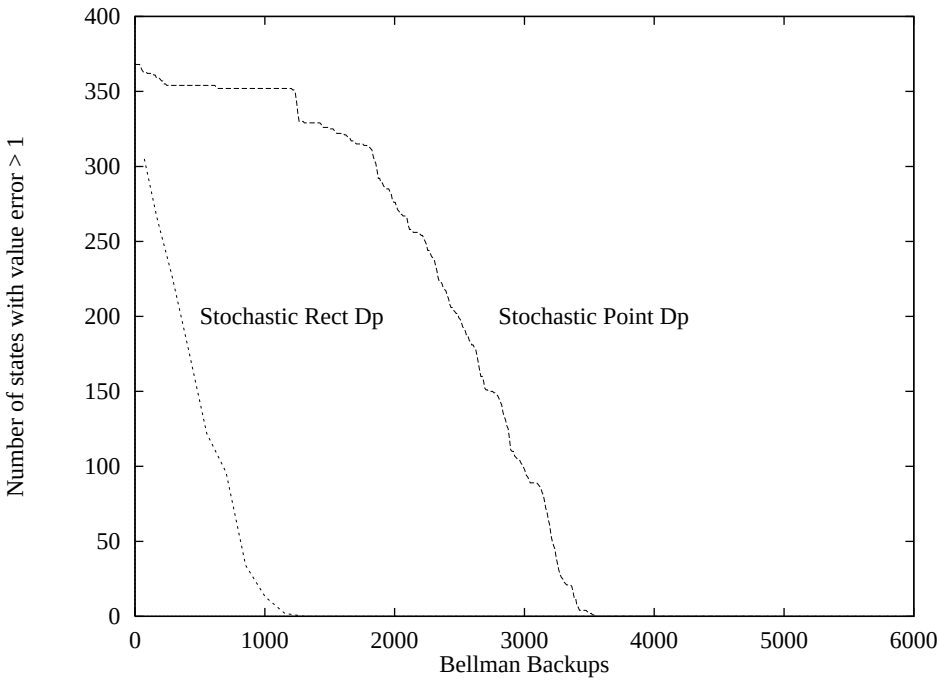

Figure 10. Value function error during dynamic programming—point-based methods versus region-based methods. Randomly-perturbed operators move in the direction opposite to the selected operator.

policy. We also measured the deviation of the current value function from the optimal value function. The results are plotted in Figures 11 and 12.

From these figures, we can see that ONLINE-RECT-DP reaches optimal performance much faster than either EBL or ONLINE-POINT-DP. Furthermore, we can see that the performance of EBL converges to a rather 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 solutions paths found in the first few trials. Some runs—with good early trials—perform much better than other runs—with very bad early trials.

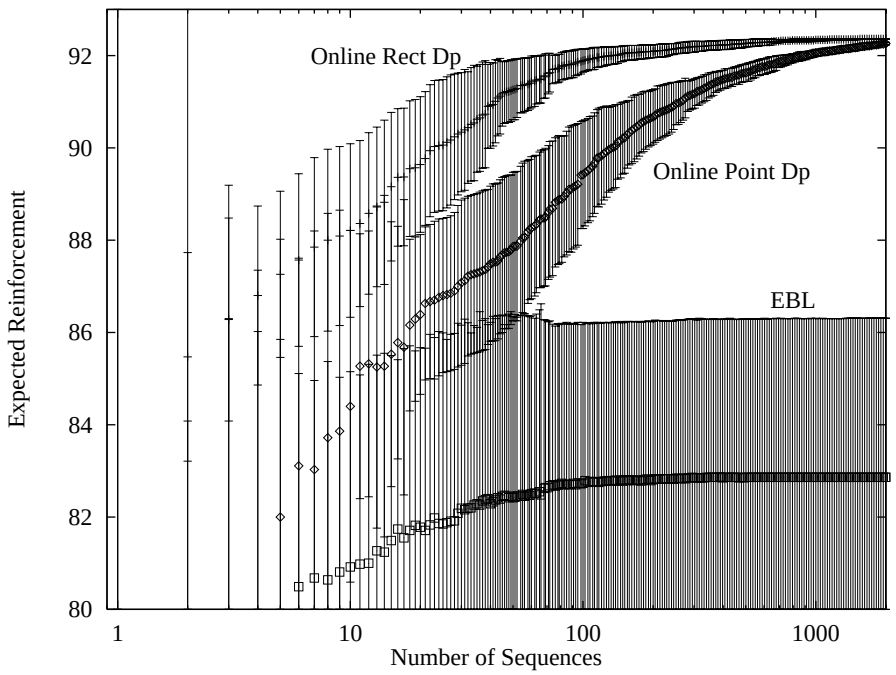

Figure 11. Comparison of expected reward for explanation-based learning (EBL), reinforcement learning (ONLINE-POINT-DP), and explanation-based reinforcement learning (ONLINE-RECT-DP). Error bars show one standard error on each side of the mean of 30 runs.

3.4. OFFLINE-RECT-DP and OFFLINE-POINT-DP in chess endgames

To demonstrate the generality of the approach, we apply the batch methods to develop optimal policies for playing chess endgames. Chess is considerably more complex than the synthetic maze task for a number of reasons. First, this domain involves counter-planning, because we have two agents with opposing goals. One player is trying to maximize its return, the other trying to minimize it. Second, although played on a two dimensional grid, applying EBRL to chess involves reasoning about multi-dimensional regions, because each abstract state has multiple objects. Finally, "state generalization" methods, referred to in Section 1.3, are very difficult apply because of the complexity of manually constructing an accurate generalization vocabulary (Quinlan, 1983; Bratko & Michie, 1980).

To incorporate the counter-planning nature of chess, we must extend the state to be a tuple describing the board position and the side that has the next move. We will refer the maximizing player as *max* and the minimizing opponent as *min.* The value of a state, $f(s, p)$ is the value of the board position s for the maximizing side

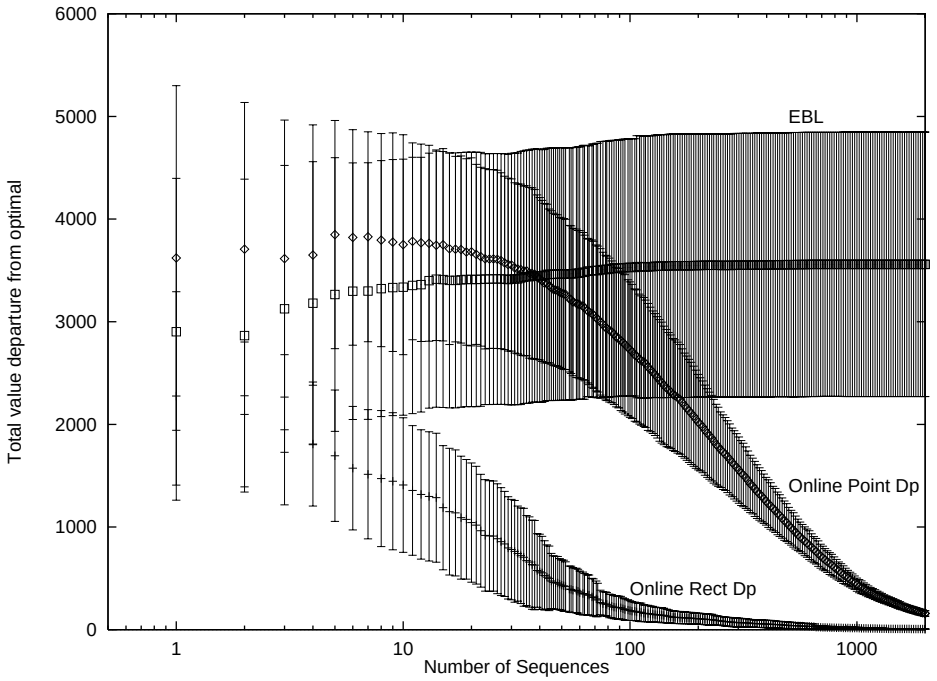

Figure 12. Comparison of summed absolute difference between the current value function and the optimal value function for three learning algorithms (EBL, ONLINE-POINT-DP, and ONLINE-RECT-DP). Error bars show one standard error on each side of the mean of 30 runs.

(p is the player who makes the next move). For goal states, the value of $f(b, p)$ is defined:

$$f(s, min) := -\infty$$
$$f(s, max) := +\infty$$

Dijkstra backups are slightly more complex. For a state where max is to move, the backups are the same as before:

$$f(s, max) := \max_{Op} \{f(Op(s), min) - cost(Op)\}$$

However, for a state where min is to move, the backups require minimization instead of maximization:

$$f(s, min) := \min_{Op} \{f(Op(s), max) + cost(Op)\}$$

Because of these minimizing backups and opposing goals, dynamic programming algorithms for counter-planning are more complicated than for ordinary planning.

In OFFLINE-POINT-DP, a state is pushed onto the priority queue whenever a better value for $f(s)$ is discovered. In counter-planning domains, if this value represents a loss, the losing player will want to wait until all possible successor moves have been backed up. Only then has the state been determined to be a loss and s can be pushed onto the priority queue. Because of the need to wait for all possible successor moves to determine correct values of $f(s)$, on-line algorithms are ineffective. In this study we implemented only the batch algorithms: OFFLINE-POINT-DP and OFFLINE-RECT-DP.

OFFLINE-POINT-DP has been applied successfully in deriving optimal policies for playing both chess and checkers endgames. In chess, Thompson (1986) has determined many 100-move or greater forced wins for five- and six-piece endings. In checkers, endgame databases up to seven pieces have significantly contributed to a program's expertise and earned it the right to play against the world champion (Schaeffer, 1991). However, since the size of the tables needed to store these policies grows exponentially with the number of pieces, OFFLINE-POINT-DP has effectively reached its limit of usefulness in these applications. A potential advantage of OFFLINE-RECT-DP is to permit deeper searches before exhausting available memory for storing the value function.

3.4.1. OFFLINE-POINT-DP *in chess endgames*

The OFFLINE-POINT-DP algorithm given in Table 1 must be modified for chess and other counter-planning domains. The main idea is still the same—the search begins with the goal states and works backwards systematically. To accommodate the two players, the backwards search is divided into maximizing and minimizing phases.

The modified OFFLINE-POINT-DP algorithm is given in Table 13. There are two queues, one called Q_{max} for storing states where *max* is to move, the other called Q_{min} for storing states where *min* is to move. The process begins by both queues being initialized with their respective goals. Then processing alternates between the two queues until both are empty.

In addition to the value function, $f(s,p)$, each state $\langle s,p\rangle$ has an associated function $g(s,p)$, which represents the number of legal moves for player p starting in state s that have not yet been backed up. Initially, $g(s,p)$ is set to the number of legal moves that player p can make in s. This function is used to determine if the value of $f(s,p)$ is indeed a loss.

The maximizing phase of the algorithm is illustrated in lines 11–25. During this phase, the backing up of values is the same as in the standard algorithm. The only difference is following the value function update. If the value determined is not a loss for *max* then the state is pushed onto the queue. However, if the value currently represents a loss, then the state is not pushed onto the queue until all forward operators have been considered (i.e., $g(s,max) = 0$). When the queue, Q_{min}, is empty, OFFLINE-POINT-DP performs the minimizing phase by backing up all states on Q_{max}. This step is illustrated in lines 26–40, and it is the exact

```
1   Let f be the value function (represented as an array
      with one element for each possible state and player).
2   f[s, min] initialized to +∞; f[s, max] initialized to −∞ for all s.
3   Let g be the "outgoing move count" (represented as an array
4     with one element for each possible board position and player).
5   For each ⟨s, p⟩ initialize g(s, p) to be the number of
      moves available to player p in state s.
6   Let Q_min and Q_max be priority queues.
7   Let G_min and G_max be the set of goal states.
8   For each g ∈ G_min, push (g, −∞) onto Q_min and set f[g, min] := −∞.
9   For each g ∈ G_max, push (g, +∞) onto Q_max and set f[g, max] := +∞.
10  While notEmpty(Q_min) ∧ notEmpty(Q_max) do begin
11    While notEmpty(Q_min) do
12          Let (s', v') := pop(Q_min).
13          For each operator Op_max do begin
14                Let P := Op_max^{-1}(s') be the pre-image of state s' for operator Op_max.
15                For each s ∈ P do begin
16                      g[s, max] := g[s, max] − 1
17                      Let v := v' − cost(Op_max) be the tentative backed-up value of state s.
18                      If v > f[s, max] then begin
19                            f[s, max] := v
20                            If f[s, max] > 0 push (s, v) onto Q_max
21                            end // if
22                      else if g[s, max] = 0 push (s, v) onto Q_max
23                      end // For s
24                end // For Op
25          end // while
26    While notEmpty(Q_max) do
27          Let (s', v') := pop(Q_max).
28          For each operator Op_min do begin
29                Let P := Op_min^{-1}(s').
30                For each s ∈ P do begin
31                      g[s, min] := g[s, min] − 1
32                      Let v := v' + cost(Op_min) be the tentative backed-up value of state s.
33                      If v < f[s, min] then begin
34                            f[s, min] := v
35                            If f[s, min] < 0 push (s, v) onto Q_min
36                            end // if v
37                      else if g[s, max] = 0 push (s, v) onto Q_min
38                      end // s
39                end // Op
40          end // while
41    end // while
```

Table 13. The OFFLINE-POINT-DP algorithm for off-line, point-based dynamic programming for counter-planning domains.

dual of the maximizing stage. The algorithm terminates when both queues become empty. With the $cost(Op)$ of both *min* and *max*'s operators being equal to 1, $f(s, p)$ represents the length of the shortest path to a win or the longest path to a loss.

```
1   Let f be the value function (represented as a collection of rectangles;
      each rectangle has an associated value).
2   Let Q_min and Q_max be priority queues.
3   Let G_min and G_max be the set of rectangles describing the goal states.
4   For each r ∈ G_min, push (r, −∞) onto Q_min and insert (r, −∞, nil) into f(min).
5   For each r ∈ G_max, push (r, +∞) onto Q_max and insert (r, +∞, nil) into f(max)
6   While notEmpty(Q_min) ∧ notEmpty(Q_max)
7     While notEmpty(Q_min) do
8           Let (r', v') := pop(Q_min).
9           For each operator Op_max do begin
10                Let P := Op_max^{-1}(r')
11                For each rectangle r ∈ P do begin
12                      Let v := v' + cost(Op_max) be the backed-up value of r.
13                      Intersect r each rectangle r_j ∈ f such that r_j ∩ r ≠ ∅
14                      Gather all of the resulting subregions of r into a set U.
15                      For each u_i ∈ U do begin
16                            Let f(u_i) be the maximum value of the rectangles intersected to form u_i.
17                            Let g(u_i) be the number of rectangles that were intersected to form u_i.
18                            If g(u_i) = the total number of possible outgoing moves for u_i
19                                  push (u_i, f(u_i)) onto Q_max.
20                                  end // if g
21                            end // for u_i
22                      insert (r, v, Op) into the collection representing f
23                      end // r
24                end // Op
25          end // while
26    While notEmpty(Q_max) do
27          Let (r', v') := pop(Q_max).
28          For each operator Op_min do begin
29                Let P := Op_min^{-1}(r')
30                For each rectangle r ∈ P do begin
31                      Let v := v' − cost(Op_min) be the backed-up value of r.
32                      Intersect r each rectangle r_j ∈ f such that r_j ∩ r ≠ ∅
33                      Gather all of the resulting subregions of r into a set U.
34                      For each u_i ∈ U do begin
35                            Let f(u_i) be the minimum value of the rectangles intersected to form u_i.
36                            Let g(u_i) be the number of rectangles that were intersected to form u_i.
37                            If g(u_i) = the total number of possible outgoing moves for u_i
38                                  push (u_i, f(u_i)) onto Q_min.
39                                  end // if g
40                            end // for u_i
41                      insert (r, v, Op) into the collection representing f
42                      end // r
43                end // Op
44          end // while
45    end // while
```

Table 14. The OFFLINE-RECT-DP algorithm for off-line, rectangle-based dynamic programming for counter-planning domains.

3.4.2. OFFLINE-RECT-DP *in chess endgames*

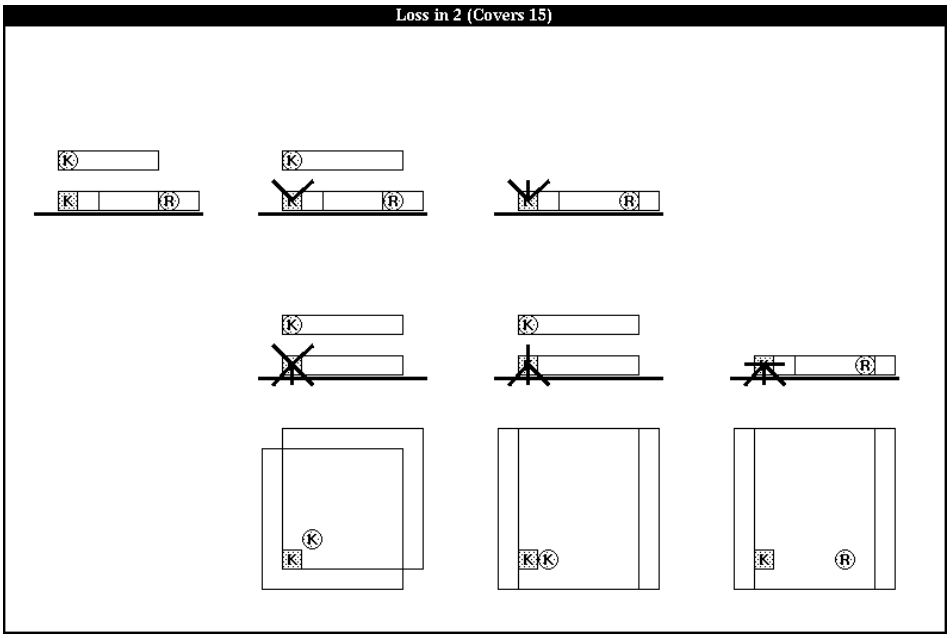

Figure 13. An example of three abstract Dijkstra backups deriving a *max*-to-play, loss-in-2-ply pattern for the king versus king-and-rook chess endgame. The loss pattern appears at upper left, and it describes a set of 15 checkmate states for *max*'s king (the square piece). Note that each piece is bounded within a rectangle. To derive this loss, preimages of *min*-to-play, win-in-1-ply patterns (illustrated in the middle row) are progressively intersected, reducing the set of *max*'s king's non-backed-up forward operators to $\emptyset$. The bottom row illustrates the provided *min*-to-play, win-in-1-ply goal patterns, where *max*'s king is immediately captured.

For chess and checkers, the algorithm for OFFLINE-RECT-DP, given in Table 2, must also be modified to employ two backup phases. However, instead of manipulating two-dimensional rectangles, our counter-planning version of OFFLINE-RECT-DP handles higher-dimensional rectangular regions. Each such region contains a two-dimensional rectangle for each playing piece on the board. To store and retrieve these regions, a multi-dimensional rectangle tree is employed (Edelsbrunner, 1983) that provides a retrieval complexity of $O(\log^{2d}(2d) + N)$, where d is the number of pieces in the region and N is the number of regions retrieved.

The modified OFFLINE-RECT-DP algorithm is given in Table 14 and begins by inserting the goal regions onto the minimizer's and maximizer's region queues, Q_{min} and Q_{max}. Then it enters a maximizing phase. This process is illustrated in Figure 13 for the king versus king-and-rook endgame.

The trickiest part of the algorithm is to determine when all outgoing moves from a region have been backed-up so that a new region can be entered into the appropriate priority queue. Preimage regions are computed as in the maze task. To backup

for the maximizing player, we generate all regions r, such that $r = Op^{-1}_{max}(r'), r' \in Q_{min}$. Next, we need to determine whether there is any subregion of r for which all outgoing moves have been backed up. To do this, we identify every region r_j in f that has a non-empty intersection with r. We intersect all such regions r_j with r, which partitions r into a set of subregions (these subregions can be cached to save subsequent recomputation). For each subregion, r_i, the number of outgoing moves that have been backed up is equal to the number of regions that were intersected to create r_i. If this number is equal to the total number of outgoing moves for r_i, then it represents a new loss region, so r_i is pushed onto *max*'s queue, Q_{max}. Finally, each r is entered into f. Note that each backed-up region r must be entered into f even if it is completely "hidden" by existing regions with better values—otherwise, we can't compute the number of outgoing moves that have been backed up.

Once Q_{min} is empty, the algorithm shifts into the minimizing phase by backing up each of the new regions on Q_{max}. The algorithm terminates when both queues are empty.

3.4.3. Experiment and results

We studied these two algorithms 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, the player with the single king. The minimizing player has both a king and a rook. The initial goal states describe *min*-to-play, win-in-one-ply positions where *max*'s king is immediately captured. There are 18,704 possible goal states for OFFLINE-POINT-DP and three goal regions for OFFLINE-RECT-DP that are initially pushed onto Q_{min}.

The results of running OFFLINE-RECT-DP and OFFLINE-POINT-DP for the entire KRK chess endgame are shown in Figure 14. A useful Dijkstra backup is one that either leads to a change in value for a region (when a shorter path to a win was identified) or reduces the number of remaining forward operators.

From Figure 14, it is clear that OFFLINE-POINT-DP requires approximately 50 times more backups than OFFLINE-RECT-DP (706,584 versus 13,619). One factor that can account for this difference is the representation of the goal states. For OFFLINE-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 a coverage of 54. OFFLINE-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. Whenever a region is backed up through a rook move, a more general region is generated. This tends to counter the specialization effect of forming loss-regions, where repeated region intersections reduce generality.

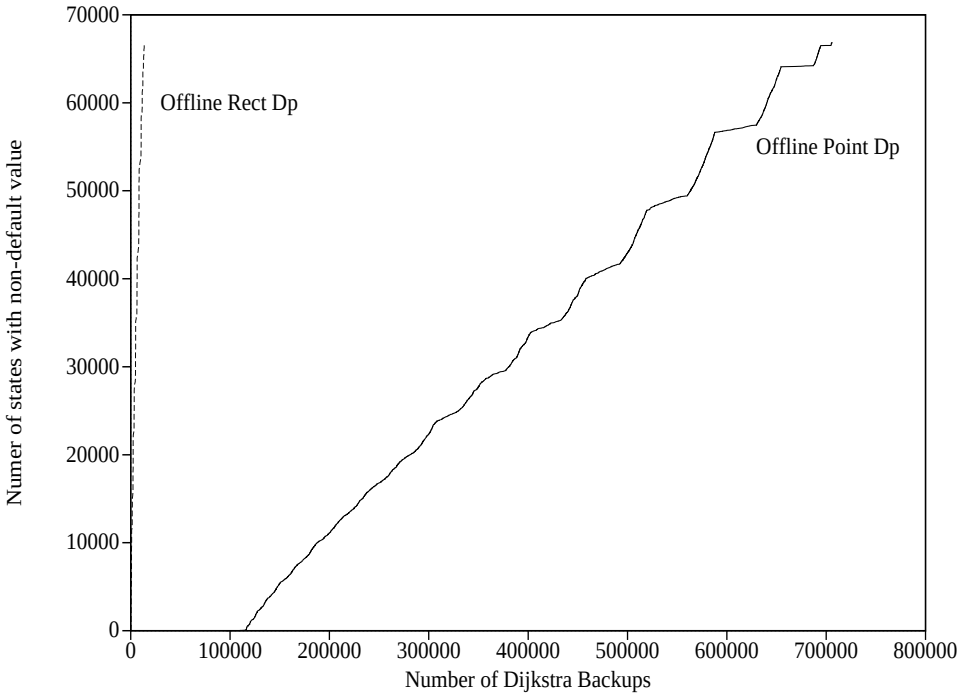

Figure 14. Coverage of batch algorithms as a function of the number of useful Dijkstra Backups for KRK chess endgame.

Another feature of the graphs is their "staircase" form, where each graph alternates between a steep and shallow slope. The steeper regions are generated during the minimizing phase where most backups result in a new win. The shallower regions are generated when the maximizing side is backing up and new loss states are only generated when the set of forward operators is reduced to $\emptyset$.

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

4. Discussion

Our current stock of algorithms for reinforcement learning and explanation-based learning suffer from many problems. The algorithms and experiments presented above provide partial solutions to some of these problems.

4.1. Implications for Reinforcement Learning

Let us begin by considering the problems confronting RL methods. Point-based dynamic programming algorithms (i.e., algorithms that store the value function as a large table) learn slowly and do not scale to large state spaces. The results in this paper show that region-based methods (Offline-Rect-Dp and Online-Rect-Dp) learn faster than point-based methods (Offline-Point-Dp and Online-Point-Dp) in deterministic and stochastic 2-D maze tasks and in chess endgames. The region-based methods achieve better coverage, better mean reward, and better approximation to the true value function in less time than the point-based methods. Region-based methods can be applied in infinite state spaces, so they provide one way of scaling up RL to solve very large problems where the state space cannot be enumerated.

Reinforcement learning algorithms that employ function approximation to represent the value function also learn slowly in many applications (e.g., Zhang & Dietterich, 1995). A possible reason for this is that general function approximators do not incorporate any domain-specific knowledge about the shape of the value function. Region-based methods can exploit prior knowledge (in the form of operator models and the reward function) to learn the value for an entire region from a single trial. Furthermore, the value function computed by region-based backups is exact, unlike the value function computed by function approximators based on state aggregation.

The effectiveness of region-based backups is limited by three factors. First, region-based backups require correct operator models. This is not a problem for speedup learning tasks, where the operators manipulate data structures inside the computer. However, in robotic and manufacturing applications where the operators manipulate an external environment, correct operator models are difficult to obtain.

The second factor is ρ—the average number of points in each region in the final value function. 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 Dijkstra backups performed by the region-based methods is reduced from the number performed by point-based methods by a factor of ρ.

The third key factor is the cost of reasoning with regions. For rectangles in two dimensions, this cost is roughly $O(\log^3(n/\rho) + \rho \log^2(n/\rho))$, which is quite reasonable. For higher dimensions, Edelsbrunner's (1983) rectangle-tree data structure requires $O(\log^{2d}(2d) + N)$ to retrieve N rectangles in d-dimensions. However, in discrete-valued spaces with many dimensions, the costs may be prohibitive (Tambe, Newell, & Rosenbloom, 1990). These high costs are the primary cause of the so-called "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).

4.2. Implications for Explanation-Based Learning

The results and perspective of this paper also provide partial solutions to many long-standing problems in explanation-based learning. In their seminal (1986) paper, Mitchell, Keller, and Kedar-Cabelli described three aspects of what they called the "imperfect theory problem": the intractable theory problem, the incomplete theory problem, and the inconsistent theory problem. Let us consider each of these in turn.

4.2.1. The Intractable Theory Problem

The intractable theory problem arises when the domain theory makes it expensive to find a "correct" explanation to which explanation-based generalization (goal regression) can be applied. This can arise for many reasons. In problems, such as those discussed in this paper, where optimal problem-solving solutions are desired, an EBL approach will only work if it is applied to an optimal operator sequence. Finding such sequences starting with a weak-method problem solver is usually intractable, so this makes it impossible to apply EBL to such problems.

The present paper has shown that EBL can be applied to these problems if we learn a value function instead of a policy. By learning a value function, a problem solving system can start with very poor problem-solving sequences and improve them incrementally. If Bellman (or Dijkstra) backups are performed in every state enough times, then the value function will converge to the optimal value function, and a one-step lookahead policy will converge to the optimal policy.

This is particularly relevant in the chess domain, where standard EBL methods are very difficult to apply. Our experiments have shown that region-based dynamic programming can be extremely valuable in chess endgame problems.

Another case in which the intractable theory problem arises is when the operators in the domain are stochastic. A traditional EBL system would observe one particular operator sequence and perform region-based backups along that sequence to learn control rules. In future problem solving trials, if a future state matched one of the learned rules, then the EBL system would try to apply the same operator sequence. However, if the initial observed sequence of states had very low probability, the resulting control rules would not yield an optimal policy. More fundamentally, EBL methods based on logical proofs lack the machinery to prove statements about the probability that a particular operator sequence will reach the goal (or about the expected return that will be received). For this reason, few people have attempted to apply EBL in stochastic domains.

The present paper has shown, however, that if we shift from learning a policy to learning a value function and if we shift from simple Dijkstra-style backups to performing Bellman backups, then stochastic domains can be handled within the same framework as deterministic domains. Hence, dynamic programming provides a solution to this form of the intractable theory problem.

However, we have seen that region-based methods are less effective in stochastic domains than in deterministic ones, because the regions are formed by the cross-

product of the preimages of all alternative outcomes of an action. If care is not taken during the dynamic programming process, the preimages can become very small, which destroys the benefits of the region-based methods. Nonetheless, in our experiments with stochastic operators, the region-based methods still had a sizable advantage over point-based methods.

There is one aspect of the intractable theory problem that region-based methods cannot address. If the search space is immense, so that it is very difficult to find any operator sequences that reach a goal state, then dynamic programming methods (either point-based or region-based) do not have much benefit. To solve such difficult problems, other techniques, particularly those based on introducing abstractions, must be applied.

4.2.2. The Incomplete Theory Problem

The incomplete theory problem arises when the problem solver does not have a complete model of its operators (or the reward function). Region-based methods cannot be applied without complete models. If operator models are incorrect, then the resulting policy will not be optimal. For example, if operators are missing preconditions, then preimages computed during 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.

The unified perspective of this paper suggests a solution: model-free RL algorithms such as Q-learning or adaptive real-time dynamic programming (ARTDP, Barto et al., 1995) can be applied in cases where nothing is known about the operators. Hence, a hybrid approach in which region-based methods are applied to states where models are available and model-free methods are applied to states where models are not available should be able to overcome the "incomplete theory problem."

More generally, a problem solver can observe the results of its interactions with the environment to learn the missing pieces of the domain theory. The results for ARTDP show that as long as the domain theory eventually converges to the correct theory, the learned policy will converge to the optimal policy.

4.2.3. The Inconsistent Theory Problem

The inconsistent theory problem arises when the domain theory permits the construction of multiple, mutually inconsisent, explanations. This can arise because of errors in formalizing the operators and their effects, but the primary source of this problem is the use of domain theories expressed in some form of non-monotonic or default logic. The ideas in this paper do not provide any insights or solutions to this problem.

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 Dijkstra backup step of dynamic programming and RL algorithms. Based on this insight, we designed and implemented batch and online algorithms that perform region-based 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.

We also compared these algorithms to standard explanation-based learning and showed that they learn optimal value functions in cases where EBL does not. Indeed, the simple step of representing the value as part of the regions computed by EBL makes it easy for EBL algorithms to find optimal policies.

Finally, we have shown how RL algorithms can address many of the open problems raised by work in explanation-based learning. Reinforcement learning methods can be applied in cases where EBL would be impossible because of incomplete or intractable domain theories.

This analysis moves us one step closer to building a unified theory of speedup learning for problem-solving systems. It provides one possible approach to improving the speed and scalability of RL algorithms, and it provides solutions to many of the long-standing problems of EBL.

Acknowledgments

T. Dietterich was supported in part by NSF grants IRI-9204129 and IRI-8657316 and by a grant from NASA Ames Research Center NAG 2-630. N. Flann was supported in part by the INEL Research Consortium. INEL is managed by Lockheed Martin Technologies Company for the U.S. Department of Energy, Idaho Operations Office, under Contract No. DE-AC07-94IDI3223. The authors gratefully acknowledge this support. We also thank the anonymous reviewers for their suggestions, which improved the paper substantially.

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.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*, 81–138.

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

Bern, M. (1990). Hidden surface removal for rectangles. *Journal of Computer and System Sciences*, *40*, 49–69.

Bertsekas, D. P., & Castanon, D. A. (1989). Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, *AC-34*, 589–598.

Bratko, I., & Michie, D. (1980). A representation for pattern-knowledge in chess endgames. In Clarke, M. R. (Ed.), *Advances in Computer Chess*, Vol. 2. Edinburgh University Press.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 726–731. Morgan Kaufmann.

Christiansen, A. D. (1992). Learning to predict in uncertain continuous tasks. In Sleeman, D., & Edwards, P. (Eds.), *Proceedings of the Ninth International Conference on Machine Learning*, pp. 72–81 San Francisco. Morgan Kaufmann.

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

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. *International Journal of Computer Mathematics*, *13*, 209–219.

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

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

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

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

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

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

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *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.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Puterman, M. L. (1994). *Markov Decision Processes.* J. Wiley & Sons, New York.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess endgames. In *Machine learning: An artificial intelligence approach*, Vol. 1, pp. 463–482. Tioga Press, Palo Alto, CA.

Sammut, C., & Cribb, J. (1990). Is learning rate a good performance criterion for learning? In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 170–178 San Francisco, CA. Morgan Kaufmann.

Schaeffer, J. (1991). Checkers program earns the right to play for world title. *Computing Research News*, *1*, 12. January.

Subramanian, D., & Feldman, R. (1990). The utility of EBL in recursive domain theories. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)* Menlo Park, CA. AAAI Press.

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

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

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

Thompson, K. (1986). Retrograde analysis of certain endgames. *ICCA Journal*, *9*(3), 131–139.

Watkins, C. J. C. (1989). *Learning from delayed rewards.* Ph.D. thesis, King's College, Cambridge.

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

Yee, R. C., Saxena, S., Utgoff, P. E., & Barto, A. G. (1990). Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eight National Conference on Artificial Intelligence (AAAI-90)*, pp. 882–888 Cambridge, MA. AAAI Press/MIT Press.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *1995 International Joint Conference on Artificial Intelligence*, pp. 1114–1120. AAAI/MIT Press, Cambridge, MA.

Received February 28, 1995
Accepted September 26, 1995
Final Manuscript March 13, 1997

Notes

1. Such operator models are also called the *domain theory*.
2. More generally, the return may be defined in many different ways including expected cumulative reward, expected cumulative discounted reward, and expected reward per unit time. See Puterman (1994) for a rigorous presentation.
3. 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.