Generalizing Dijkstra's Algorithm and Gaussian Elimination for Solving MDPs

H. Brendan McMahan **Geoffrey J. Gordon**

May 2005
CMU-CS-05-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We study the problem of computing the optimal value function for a Markov decision process with positive costs. Computing this function quickly and accurately is a basic step in many schemes for deciding how to act in stochastic environments. There are efficient algorithms which compute value functions for special types of MDPs: for deterministic MDPs with S states and A actions, Dijkstra's algorithm runs in time $O(AS \log S)$. And, in single-action MDPs (Markov chains), standard linear-algebraic algorithms find the value function in time $O(S^3)$, or faster by taking advantage of sparsity or good conditioning. Algorithms for solving general MDPs can take much longer: we are not aware of any speed guarantees better than those for comparably-sized linear programs. We present a family of algorithms which reduce to Dijkstra's algorithm when applied to deterministic MDPs, and to standard techniques for solving linear equations when applied to Markov chains. More importantly, we demonstrate experimentally that these algorithms perform well when applied to MDPs which "almost" have the required special structure.

This work was funded in part by DARPA's MICA project, AFRL contract F30602-01-C-0219.

Keywords: Planning under uncertainty, Markov Decision Processes, Prioritized Sweeping, Dijsktra's Algorithm, Gaussian Elimination, Value Iteration, Improved Prioritized Sweeping (IPS), Prioritized Policy Iteration (PPI), Gauss-Dijsktra Elimination (GDE).

1 Introduction

We consider the problem of finding an optimal policy in a Markov decision process with non-negative costs and a zero-cost, absorbing goal state. This problem is sometimes called the stochastic shortest path problem. Let V^* be the optimal state value function, and let Q^* be the optimal state-action value function. That is, let $V^*(x)$ be the expected cost to reach the goal when starting at state x and following the best possible policy, and let $Q^*(x, a)$ be the same except that the first action must be a. At all non-goal states x and all actions a, V^* and Q^* satisfy *Bellman's equations*:

$$\begin{aligned} V^*(x) &= \min_{a \in A} Q^*(x, a) \\ Q^*(x, a) &= c(x, a) + \sum_{y \in \mathrm{succ}(x,a)} P(y \mid x, a) V^*(y) \end{aligned}$$

where A is the set of legal actions, $c(x, a)$ is the expected cost of executing action a from state x, and $P(y \mid x, a)$ is the probability of reaching state y when executing action a from state x. The set $\mathrm{succ}(x, a)$ contains all possible possible successors of state x under action a, except that the goal state is always excluded.[1]

Many algorithms for planning in Markov decision processes work by maintaining estimates V and Q of V^* and Q^*, and repeatedly updating the estimates to reduce the difference between the two sides of the Bellman equations (called the *Bellman error*). For example, value iteration (VI) repeatedly loops through all states x performing *backup* operations at each one:

for all actions a
  $Q(x, a) \leftarrow c(x, a) + \sum_{y \in \mathrm{succ}(x,a)} P(y \mid x, a) V(y)$
end for
$V(x) \leftarrow \min_{a \in A} Q(x, a)$

On the other hand, Dijkstra's algorithm uses *expansion* operations at each state x instead:

$V(x) \leftarrow \min_{a \in A} Q(x, a)$
for all $(y, b) \in \mathrm{pred}(x)$
  $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \mathrm{succ}(y,b)} P(x' \mid y, b) V(x')$
end for

Here $\mathrm{pred}(x)$ is the set of all state-action pairs (y, b) such that taking action b from state y has a positive chance of reaching state x. For good recent references on value iteration and Dijkstra's algorithm, see [4] and [7].

Any sequence of backups or expansions is guaranteed to make V and Q converge to the optimal V^* and Q^* so long as we visit each state infinitely often. Of course, some sequences will converge much more quickly than others. A wide variety of algorithms have attempted to find good state-visitation orders to ensure fast convergence. For example, Dijkstra's algorithm is guaranteed to find an optimal ordering for a deterministic positive-cost MDP; for general MDPs, algorithms like prioritized sweeping [17], generalized

[1] To simplify notation, we have omitted the possibility of discounting. A discount γ can be simulated by reducing $P(y \mid x, a)$ by a factor of γ for all $y \neq$ goal and increasing $P(\text{goal} \mid x, a)$ accordingly. We assume that V^* and Q^* are well-defined, *i.e.*, that no state has infinite $V^*(x)$.

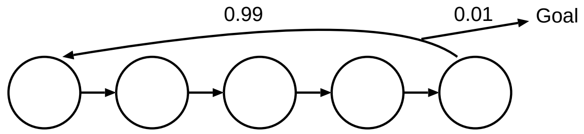

Figure 1: A Markov chain for which backup-based methods converge slowly. Each action costs 1.

prioritized sweeping [1], RTDP [3], LRTDP [6], Focussed Dynamic Programming [12], and HDP [5] all attempt to compute good orderings.

Algorithms based on backups or expansions have an important disadvantage, though: they can be slow at policy evaluation in MDPs with even a few stochastic transitions. For example, in the Markov chain of Figure 1 (which has only one stochastic transition), the best possible ordering for value iteration will only reduce Bellman error by 1% with each five backups. To find the optimal value function quickly for this chain (or for an MDP which contains it), we turn instead to methods which solve systems of linear equations.

The policy iteration algorithm alternates between steps of *policy evaluation* and *policy improvement*. If we fix an arbitrary policy and temporarily ignore all off-policy actions, the Bellman equations become linear. We can solve this set of linear equations to evaluate our policy, and set V to be the resulting value function. Given V, we can compute a *greedy policy* π under V, given by $\pi(x) = \arg\min_a Q(x, a)$. By fixing a greedy policy we get another set of linear equations, which we can also solve to compute an even better policy. Policy iteration is guaranteed to converge so long as the initial policy has a finite value function. Within the policy evaluation step of policy iteration methods, we can choose any of several ways to solve our set of linear equations [18]. For example, we can use Gaussian elimination, sparse Gaussian elimination, or biconjugate gradients with any of a variety of preconditioners. We can even use value iteration, although as mentioned above value iteration may be a slow way to solve the Bellman equations when we are evaluating a fixed policy.

Of the algorithms discussed above, no single one is fast at solving all types of Markov decision process. Backup-based and expansion-based methods work well when the MDP has short or nearly deterministic paths without much chance of cycles, but can converge slowly in the presence of noise and cycles. On the other hand, policy iteration evaluates each policy quickly, but may spend work evaluating a policy even after it has become obvious that another policy is better.

This paper describes three new algorithms which blend features of Dijkstra's algorithm, value iteration, and policy iteration. In Section 2, we describe Improved Prioritized Sweeping. IPS reduces to Dijkstra's algorithm when given a deterministic MDP, but also works well on MDPs with stochastic outcomes. In Section 3, we develop Prioritized Policy Iteration, by extending IPS by incorporating policy evaluation steps. Section 4 describes Gauss-Dijkstra Elimination (GDE), which interleaves policy evaluation and prioritized scheduling more tightly. GDE reduces to Dijkstra's algorithm for deterministic MDPs, and to Gaussian elimination for policy evaluation. In Section 6, we experimentally demonstrate that these algorithms extend the advantages of Dijkstra's algorithm to "mostly" deterministic MDPs, and that the policy evaluation performed by PPI and GDE speeds convergence on problems where backups alone would be slow.

main
queue.clear()
$(\forall x)$ closed$(x) \leftarrow$ false
$(\forall x)$ $V(x) \leftarrow M$
$(\forall x, a)$ $Q(x, a) \leftarrow M$
$(\forall a)$ $Q(\text{goal}, a) \leftarrow 0$
closed(goal) $\leftarrow$ true
$(\forall x)$ $\pi(x) \leftarrow$ undefined
π(goal) = arbitrary
update(goal)
while (not queue.isempty())
    $x \leftarrow$ queue.pop()
    closed$(x) \leftarrow$ true
    update(x)
end while

update(x)
$V(x) \leftarrow Q(x, \pi(x))$
for all $(y, b) \in \text{pred}(x)$
    $Q_{\text{old}} \leftarrow Q(y, \pi(y))$ (or M if $\pi(y)$ undefined)
    $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \text{succ}(y,b)} P(x' \mid y, b) V(x')$
    if ((not closed(y)) and $Q(y, b) < Q_{\text{old}}$))
        $\text{pri} \leftarrow Q(y, b)$ $(*)$
        $\pi(y) \leftarrow b$
        queue.decreasepriority(y, pri)
    end if
end for

Figure 2: Dijkstra's algorithm, in a notation which will allow us to generalize it to stochastic MDPs. The variable "queue" is a priority queue which returns the smallest of its elements each time it is popped. The constant M is an arbitrary very large positive number.

2 Improved Prioritized Sweeping

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is shown in Figure 2. Its basic idea is to keep states on a priority queue, sorted by how urgent it is to expand them. The priority queue is assumed to support operations queue.pop(), which removes and returns the queue element with numerically lowest priority; queue.decreasepriority(x, p), which puts x on the queue if it wasn't there, or if it was there with priority $> p$ sets its priority to p, or if it was there with priority $< p$ does nothing; and queue.clear(), which empties the queue.

In deterministic Markov decision processes with positive costs, it is always possible to find a new state x to expand whose value we can set to $V^*(x)$ immediately. So, in these MDPs, Dijkstra's algorithm touches each state only once while computing V^*, and is therefore by far the fastest way to find a complete policy.

In MDPs with stochastic outcomes for some actions, it is in general impossible to efficiently compute an optimal order for expanding states. An optimal order is one for which we can always determine $V^*(x)$ using only $V^*(y)$ for states y which come before x in the ordering. Even if there exists such an ordering (*i.e.*, if there is an acyclic optimal policy), we might need to look at non-local properties of states to find it: Figure 3 shows an MDP with four non-goal states (numbered 1–4) and two actions (a and b). In this MDP, the optimal policy is acyclic with ordering $G3214$. But, after expanding the goal state, there is no way to tell which of states 1 and 3 to expand next: both have one deterministic action which reaches the goal, and one stochastic action that reaches the goal half the time and an unexplored state half the time. If we expand either one we will set its policy to action a and its value to 10; if we happen to choose state 3 we will be correct, but the optimal action from state 1 is b and $V^*(1) = 13/2 < 10$.

Several algorithms, most notably prioritized sweeping [17] and generalized prioritized sweeping [1],

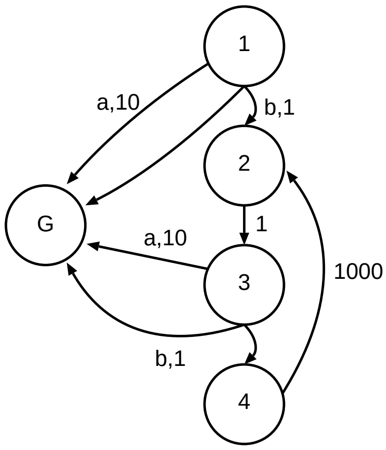

Figure 3: An MDP whose best state ordering is impossible to determine using only local properties of the states. Arcs which split correspond to actions with stochastic outcomes; for example, taking action b from state 1 reaches G with probability 0.5 and 2 with probability 0.5.

have attempted to extend the priority queue idea to MDPs with stochastic outcomes. These algorithms give up the property of visiting each state only once in exchange for solving a larger class of MDPs. However, neither of these algorithms reduce to Dijkstra's algorithm if the input MDP happens to be deterministic. Therefore, they potentially take far longer to solve a deterministic or nearly-deterministic MDP than they need to. In the next section, we discuss what properties an expansion-scheduling algorithm needs to have to reduce to Dijkstra's algorithm on deterministic MDPs.

2.2 Generalizing Dijkstra

We will consider algorithms which replace the line $(*)$ in Figure 2 by other priority calculations that maintain the property that when the input MDP is deterministic with positive edge costs an optimal ordering is produced. If the input MDP is stochastic, a single pass of a generalized Dijkstra algorithm generally will not compute V^*, so we will have to run multiple passes. Each subsequent pass can start from the value function computed by the previous pass (instead of from $V(x) = M$ like the first pass), so multiple passes will cause V to converge to V^*. (Likewise, we can save Q values from pass to pass.) We now consider several priority calculations that satisfy the desired property.

Large Change in Value The simplest statistic which allows us to identify completely-determined states, and the one most similar in spirit to prioritized sweeping, is how much the state's value will change when we expand it. In line $(*)$ of Figure 2, suppose that we set

(1) $$\text{pri} \leftarrow d(V(y) - Q(y, b))$$

for some monotone decreasing function $d(\cdot)$. Any state y with $\text{closed}(y) = \text{false}$ (called an open state) will have $V(y) = M$ in the first pass, while closed states will have lower values of $V(y)$. So, any deterministic action leading to a closed state will have lower $Q(y, b)$ than any action which might lead to an open state. And, any open state y which has a deterministic action b leading to a closed state will be on our queue with priority at most $d(V(y) - Q(y, b)) = d(M - Q(y, b))$. So, if our MDP contains only deterministic actions, the state at the head of the queue will the open state with the smallest $Q(y, b)$—identical to Dijkstra's algorithm.

Note that prioritized sweeping and generalized prioritized sweeping perform backups rather than expansions, and use a different estimates of how much a state's value will change when updated. Namely, they keep track of how much a state's successors' values have changed and base their priorities on these changes weighted by the corresponding transition probabilities. This approach, while in the spirit of Dijkstra's algorithm, does not reduce to Dijkstra's algorithm when applied to deterministic MDPs. Wiering ([19]) discusses the priority function (1), but he does not prescribe the uniform pessimistic initialization of the value function which is given in Figure 2. This pessimistic initialization is necessary to make (1) reduce to Dijkstra's algorithm. Other authors (for example Dietterich and Flann ([10])) have discussed pessimistic initialization for prioritized sweeping, but only in the context of the original non-Dijkstra priority scheme for that algorithm.

One problem with the priority scheme of equation (1) is that it only reduces to Dijkstra's algorithm if we uniformly initialize $V(x) \leftarrow M$ for all x. If instead we pass in some nonuniform $V(x) \geq V^*(x)$ (such as one which we computed in a previous pass of our algorithm, or one we got by evaluating a policy provided by a domain expert), we may not expand states in the correct order in a deterministic MDP.[2] This property is somewhat unfortunate: by providing stronger initial bounds, we may cause our algorithm to run longer. So, in the next few subsections we will investigate additional priority schemes which can help alleviate this problem.

Low Upper Bound on Value Another statistic which allows us to identify completely-determined states x in Dijkstra's algorithm is an upper bound on $V^*(x)$. If, in line $(*)$ of Figure 2, we set

$$\text{pri} \leftarrow m(Q(y, b)) \tag{2}$$

for some monotone increasing function $m(\cdot)$, then any open state y which has a deterministic action b leading to a closed state will be on our queue with priority at most $m(Q(y, b))$. (Note that $Q(y, b)$ is an upper bound on $V^*(y)$ because we have initialized $V(x) \leftarrow M$ for all x.) As before, in a deterministic MDP, the head of the queue will be the open state with smallest $Q(y, b)$. But, unlike before, this fact holds no matter how we initialize V (so long as $V(x) > V^*(x)$): in a deterministic positive-cost MDP, it is always safe to expand the open state with the lowest upper bound on its value.

High Probability of Reaching Goal Dijkstra's algorithm can also be viewed as building a set of closed states, whose V^* values are completely known, by starting from the goal state and expanding outward. According to this intuition, we should consider maintaining an estimate of how well-known the values of our states are, and adding the best-known states to our closed set first.

For this purpose, we can add extra variables $p_{\text{goal}}(x, a)$ for all states x and actions a, initialized to 0 if x is a non-goal state and 1 if x is a goal state. Let us also add variables $p_{\text{goal}}(x)$ for all states x, again initialized to 0 if x is a non-goal state and 1 if x is a goal state.

To maintain the p_{goal} variables, each time we update $Q(y, b)$ we can set

$$p_{\text{goal}}(y, b) \leftarrow \sum_{x' \in \text{succ}(y,b)} P(x' \mid y, b) p_{\text{goal}}(x')$$

[2]We need to be careful passing in arbitrary $V(x)$ vectors for initialization: if there are any optimal but underconsistent states (states whose $V(x)$ is already equal to $V^*(x)$, but whose $V(x)$ is less than the right-hand side of the Bellman equation), then the check $Q(y, b) < V(y)$ will prevent us from pushing them on the queue even though their predecessors may be inconsistent. So, such an initialization for V may cause our algorithm to terminate prematurely before $V = V^*$ everywhere. Fortunately, if we initialize using a V computed from a previous pass of our algorithm, or set V to the value of some policy, then there will be no optimal but underconsistent states, so this problem will not arise.

And, when we assign $V(x) \leftarrow Q(x,a)$ we can set

$$p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x,a)$$

(in this case, we will call a the *selected action* from x). With these definitions, $p_{\text{goal}}(x)$ will always remain equal to the probability of reaching the goal from x by following selected actions and at each step moving from a state expanded later to one expanded earlier (we call such a path a *decreasing path*). In other words, $p_{\text{goal}}(x)$ tells us what fraction of our current estimate $V(x)$ is based on fully-examined paths which reach the goal.

In a deterministic MDP, p_{goal} will always be either 0 or 1: it will be 0 for open states, and 1 for closed states. Since Dijkstra's algorithm never expands a closed state, we can combine any decreasing function of $p_{\text{goal}}(x)$ with any of the above priority functions without losing our equivalence to Dijkstra. For example, we could use

$$\text{pri} \leftarrow m(Q(y,b), 1 - p_{\text{goal}}(y)) \tag{3}$$

where m is a two-argument monotone function.[3]

In the first sweep after we initialize $V(x) \leftarrow M$, priority scheme (3) is essentially equivalent to schemes (1) and (2): the value $Q(x,a)$ can be split up as

$$p_{\text{goal}}(x,a)Q_D(x,a) + (1 - p_{\text{goal}}(x,a))M$$

where $Q_D(x,a)$ is the expected cost to reach the goal assuming that we follow a decreasing path. That means that a fraction $1 - p_{\text{goal}}(x,a)$ of the value $Q(x,a)$ will be determined by the large constant M, so state-action pairs with higher $p_{\text{goal}}(x,a)$ values will almost always have lower $Q(x,a)$ values. However, if we have initialized $V(x)$ in some other way, then equation (1) no longer reduces to Dijkstra's algorithm, while equations (2) and (3) are different but both reduce to Dijkstra's algorithm on deterministic MDPs.

All of the Above Instead of restricting ourselves to just one of the priority functions mentioned above, we can combine all of them: since the best states to expand in a deterministic MDP will win on any one of the above criteria, we can use any monotone function of all of the criteria and still behave like Dijkstra in deterministic MDPs. For example, we can take the sum of two of the priority functions, or the product of two positive priority functions; or, we can use one of the priorities as the primary sort key and break ties according to a different one.

We have experimented with several different combinations of priority functions; the experimental results we report use the priority functions

$$\text{pri}_1(x,a) = \frac{Q(x,a) - V(x)}{Q(x,a) + 1} \tag{4}$$

and

$$\text{pri}_2(x,a) = \langle 1 - p_{\text{goal}}(x),\ \text{pri}_1(x,a) \rangle \tag{5}$$

The pri_1 function combines the value change criterion (1) with the upper bound criterion (2). It is always negative or zero, since $0 < Q(x,a) \leq V(x)$. It decreases when the value change increases (since $1/Q(x,a)$ is positive), and it increases as the upper bound increases (since $1/x$ is a monotone decreasing function when $x > 0$, and since $Q(x,a) - V(x) \leq 0$).

The pri_2 function uses p_{goal} as a primary sort key and breaks ties according to pri_1. That is, pri_2 returns a vector in $\mathbb{R}^2$ which should be compared according to lexical ordering (*e.g.*, $(3,3) < (4,2) < (4,3)$).

[3] A monotone function with multiple arguments is one which always increases when we increase one of the arguments while holding the others fixed.

update(x)
$V(x) \leftarrow Q(x, \pi(x))$
for all $(y, b) \in \mathrm{pred}(x)$
  $Q_{\text{old}} \leftarrow Q(y, \pi(y))$ (or M if $\pi(y)$ undefined)
  $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \mathrm{succ}(y,b)} P(x' \mid y, b) Q(x', \pi(x'))$
  if $(Q(y, b) < Q_{\text{old}})$
    $\mathrm{pri} \leftarrow (Q(y, b) - V(y))/(Q(y, b) + 1)$
    $\pi(y) \leftarrow b$
    if $(|V(y) - Q(y, b)| > \epsilon)$
      queue.decreasepriority(y, pri)
    end if
  end if
end for

Figure 4: The **update** function for the Improved Prioritized Sweeping algorithm. The **main** function is the same as for Dijkstra's algorithm. As before, "queue" is a priority min-queue and M is a very large positive number.

2.3 Sweeps vs. Multiple Updates

The algorithms we have described so far in this section must update every state once before updating any state twice. We can also consider a version of the algorithm which does not enforce this restriction; this multiple-update algorithm simply skips the check "if not closed(y)" which ensures that we don't push a previously-closed state onto the priority queue. The multiple-update algorithm still reduces to Dijkstra's algorithm when applied to a deterministic MDP: any state which is already closed will fail the check $Q(y, b) < V(y)$ for all subsequent attempts to place it on the priority queue.

Experimentally, the multiple-update algorithm is faster than the algorithm which must sweep through every state once before revisiting any state. Intuitively, the sweeping algorithm can waste a lot of work at states far from the goal before it determines the optimal values of states near the goal.

In the multiple-update algorithm we are always effectively in our "first sweep," and so since we initialize uniformly to a large constant M we can reduce to Dijkstra's algorithm by using priority pri_1 from equation (4). The resulting algorithm is called Improved Prioritized Sweeping; its update method is listed in Figure 4.

We have not mentioned how one should check for convergence when a generalized Dijkstra algorithm is used on an arbitrary MDP. As is typical for value-function based methods, we declare convergence when the maximum Bellman error (over all states) drops below some preset limit ϵ. This is implemented in IPS by an extra check that ensures all states on the priority queue have Bellman error at least ϵ; when the queue is empty it is easy to show that no such states remain. Similar methods are used for our other algorithms.

```
main()
(∀x) V(x) ← M, V_old(x) ← M
V(goal) ← 0, V_old(goal) ← 0
while (true)
   (∀x) π(x) ← undefined
   Δ ← 0
   sweep()
   if (Δ < tolerance)
      declare convergence
   end if
   (∀x) V_old(x) ← V(x)
   V ← evaluate policy π(x)
end while

sweep()
(∀x) closed(x) ← false
(∀x) p_goal(x) ← 0
closed(goal) ← true
update(goal)
while (not queue.isempty())
   x ← queue.pop()
   closed(x) ← true
   update(x)
end while

update(x)
for all (y, a) ∈ pred(x)
   if (closed(y))
      Q(y, a) ← c(y, a) + Σ_{x'∈succ(y,a)} P(x' | y, a)V(x')
      Δ ← max(Δ, V(y) − Q(y, a))
   else
      for all actions b
         Q_old ← Q(y, π(y))  (or M if π(y) undefined)
         Q(y, b) ← c(y, b) + Σ_{x'∈succ(y,b)} P(x' | y, b)V(x')
         p_goal(y, b) ← Σ_{x'∈succ(y,b)} P(x' | y, b)p_goal(x')
                          + P(goal | y, b)
         if (Q(y, b) < Q_old)
            V(y) ← Q(y, b)
            π(y) ← b
            p_goal(y) ← p_goal(y, b)
            pri ← ⟨1 − p_goal(x), (V(y) − V_old(y))/V(y)⟩
            queue.decreasepriority(y, pri)
         end if
      end for
   end if
end for
```

Figure 5: The Prioritized Policy Iteration algorithm. As before, "queue" is a priority min-queue and M is a very large positive number.

3 Prioritized Policy Iteration

The Improved Prioritized Sweeping algorithm works well on MDPs which are moderately close to being deterministic. Once we start to see large groups of states with strongly interdependent values, there will be no expansion order which will allow us to find a good approximation to V^* in a small number of visits to each state. The MDP of Figure 1 is an example of this problem: because there is a cycle which has high probability and visits a significant fraction of the states, the values of the states along the cycle depend strongly on each other.

To avoid having to expand states repeatedly to incorporate the effect of cycles, we will turn to algorithms that occasionally do some work to evaluate the current policy. When they do so, they will temporarily fix the current actions to make the value determination problem linear. The simplest such algorithm is policy iteration, which alternates between complete policy evaluation (which solves an $S \times S$ system of linear equations in an S-state MDP) and greedy policy improvement (which picks the action which achieves the minimum on the right-hand side of Bellman's equation at each state).

We will describe two algorithms which build on policy iteration. The first algorithm, called Prioritized Policy Iteration, is the subject of the current section. PPI attempts to improve on policy iteration's greedy policy improvement step, doing a small amount of extra work during this step to try to reduce the number of policy evaluation steps. Since policy evaluation is usually much more expensive than policy improvement, any reduction in the number of evaluation steps will usually result in a better total planning time. The second algorithm, which we will describe in the Section 4, tries to interleave policy evaluation and policy improvement on a finer scale to provide more accurate Q and p_{goal} estimates for picking actions and calculating priorities on the fringe.

Pseudo-code for PPI is given in Figure 5. The main loop is identical to regular policy iteration, except for a call to `sweep()` rather than to a greedy policy improvement routine. The policy evaluation step can be implemented efficiently by a call to a low-level matrix solver; such a low-level solver can take advantage of sparsity in the transition dynamics by constructing an explicit LU factorization [11], or it can take advantage of good conditioning by using an iterative method such as stabilized biconjugate gradients [2]. In either case, we can expect to be able to evaluate policies efficiently even in large Markov decision processes.

The policy improvement step is where we hope to beat policy iteration. By performing a prioritized sweep through state space, so that we examine states near the goal before states farther away, we can base many of our policy decisions on multiple steps of look-ahead. Scheduling the expansions in our sweep according to one of the priority functions previously discussed insures PPI reduces to Dijkstra's algorithm: when we run it on a deterministic MDP, the first sweep will compute an optimal policy and value function, and will never encounter a Bellman error in a closed state. So Δ will be 0 at the end of the sweep, and we will pass the convergence test before evaluating a single policy. On the other hand, if there are no action choices then PPI will not be much more expensive than solving a single set of linear equations: the only additional expense will be the cost of the sweep. If B is a bound on the number of outcomes of any action, then this cost is $O((BA)^2 S \log S)$, typically much less expensive than solving the linear equations (assuming $B, A << S$). For PPI, we chose to use the pri_2 schedule from equation (5). Unlike pri_1 (equation (4)), pri_2 forces us to expand states with high p_{goal} first, even when we have initialized V to the value of a near-optimal policy.

In order to guarantee convergence, we need to set $\pi(x)$ to a greedy action with respect to V before each policy evaluation. Thus in the **update**(x) method of PPI, for each state y for which there exists some action that reaches x, we re-calculate $Q(y, b)$ values for all actions b. In IPS, we only calculated $Q(y, b)$ for actions b that reach x. The extra work is necessary in PPI because the stored Q values may be unrelated to the current V (which was updated by policy evaluation), and so otherwise $\pi(x)$ might not be set to a greedy action. Other Q-value update schemes are possible,[4] and will lead to convergence as long as they fix a greedy policy. Note also that extra work is done if the loops in **update** are structured as in Figure 5; with a slight reduction in clarity, they can be arranged so that each predecessor state y is backed up only once.

One important additional tweak to PPI is to perform multiple sweeps between policy evaluation steps. Since policy evaluation tends to be more expensive, this allows a better tradeoff to be made between evaluation and improvement via expansions.

In future work we intend to investigate whether we can find better policies more efficiently by allowing ourselves to update some states multiple times while visiting others only once. Directly plugging in a multi-update algorithm such as Improved Prioritized Sweeping is unlikely to provide a speedup, since such an

[4]For example, we experimented with only updating $Q(y, b)$ when $P(x \mid y, b) > 0$ in **update** and then doing a single full backup of each state after popping it from the queue, ensuring a greedy policy. This approach was on average slower than the one presented above.

algorithm may do a lot of work before it visits all states once; but, it may be advantageous to run a single sweep, save the list of inconsistent states encountered during that sweep, and then start an algorithm like IPS from that list.

Another possible optimization which we hope to try is to restrict some policy evaluations to a subset of the states. By fixing a reduced set of states (called an envelope [9]) which contains mostly "important" states, we can hope to gain most of the benefits of policy evaluation at a fraction of the cost. There are many ways to pick an envelope; for example, the LAO* algorithm [14] is one popular one.

4 Gauss-Dijkstra Elimination

The Gauss-Dijkstra Elimination algorithm continues the theme of taking advantage of both Dijkstra's algorithm and efficient policy evaluation, but it interleaves them at a deeper level.

Gaussian Elimination and MDPs Fixing a policy π for an MDP produces a Markov chain and a vector of costs c. If our MDP has S states (not including the goal state), let P^π be the $S \times S$ matrix with entries $P^\pi_{xy} = P(y \mid x, \pi(x))$ for all $x, y \neq \text{goal}$. Finding the values of the MDP under the given policy reduces to solving the linear equations

$$(I - P^\pi)V = c$$

To solve these equations, we can run Gaussian elimination and backsubstitution on the matrix $(I - P^\pi)$. Gaussian elimination calls **rowEliminate**(x) (defined in Figure 6, where Θ is initialized to P^π and w to c) for all x from 1 to S in order,[5] zeroing out the subdiagonal elements of $(I - P^\pi)$. Backsubstitution calls **backsubstitute**(x) for all x from S down to 1 to compute $(I - P^\pi)^{-1}c$. In Figure 6, $\Theta_{x\cdot}$ denotes the x'th row of Θ, and $\Theta_{y\cdot}$ denotes the y'th row. We show updates to $p_{\text{goal}}(x)$ explicitly, but it is easy to implement these updates as an extra dense column in Θ.

To see why Gaussian elimination works faster than Bellman backups in MDPs with cycles, consider again the Markov chain of Figure 1. While value iteration reduces Bellman error by only 1% per sweep on this chain, Gaussian elimination solves it exactly in a single sweep. The starting $(I - P^\pi)$ matrix and c vector are:

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & -0.99 & -0.01 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right], \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

(for clarity, we have shown $-p_{\text{goal}}(x)$ as an additional column separated by a bar). The first call to **rowEliminate** changes row 2 to:

$$\left[\begin{array}{ccccc|c} 0 & 1 & 0 & 0 & -0.99 & -0.01 \end{array}\right], \begin{bmatrix} 100 \end{bmatrix}$$

We can interpret this modified row 2 as a macro-action: we start from state 2 and execute our policy until we reach a state other than 1 or 2. (In this case, we will end up at the goal with probability 0.01 and in state 5 with probability 0.99.) Each subsequent call to **rowEliminate** zeros out one of the -1s below the diagonal and defines another macro-action of the form "start in state i and execute until we reach a state other than 1

[5] Using the Θ representation causes a few minor changes to the Gaussian elimination code, but it has the advantage that (Θ, w) can always be interpreted as a Markov chain which is has the same value function as the original (P^π, c). Also, for simplicity we will not consider pivoting; if π is a proper policy then $(I - \Theta)$ will always have a nonzero entry on the diagonal.

through i.” After four calls we are left with

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & -0.99 & -0.01 \\ 0 & 1 & 0 & 0 & -0.99 & -0.01 \\ 0 & 0 & 1 & 0 & -0.99 & -0.01 \\ 0 & 0 & 0 & 1 & -0.99 & -0.01 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right], \left[\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 1 \end{array}\right]$$

The last call to **rowEliminate** zeros out the last subdiagonal element (in line (1)), setting row 5 to:

$$\left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 0.01 & -0.01 \end{array}\right], \left[5\right] \tag{6}$$

Then it divides the whole row by 0.01 (line (2)) to get:

$$\left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right], \left[500\right] \tag{7}$$

The division accounts for the fact that we may visit state 5 multiple times before our macro-action terminates: equation (6) describes a macro-action which has a 99% chance of self-looping and ending up back in state 5, while equation (7) describes the macro-action which keeps going after a self-loop (an average of 100 times) and only stops when it reaches the goal.

At this point we have defined a macro-action for each state which is guaranteed to reach either a higher-numbered state or the goal. We can immediately determine that $V^*(5) = 500$, since its macro-action always reaches the goal directly. Knowing the value of state 5 lets us determine $V^*(4)$, and so forth: each call to **backsubstitute** tells us the value of at least one additional state.

Note that there are several possible ways to arrange the elimination computations in Gaussian elimination. Our example shows *row Gaussian elimination*,[6] in which we eliminate the first $k-1$ elements of row k by using rows 1 through $k-1$; the advantage of using this ordering for GDE is that we need not fix an action for state x until we pop it from the priority queue and eliminate its row.

Gauss-Dijkstra Elimination Gauss-Dijkstra elimination combines the above Gaussian elimination process with a Dijkstra-style priority queue that determines the order in which states are selected for elimination. The main loop is the same as the one for PPI, except that the policy evaluation call is removed and **sweep**() is replaced by **GaussDijkstraSweep**(). Pseudo-code for **GaussDijkstraSweep**() is given in Figure 6.

When x is popped from the queue, its action is fixed to a greedy action. The outcome distribution for this action is used to initialize $\Theta_{x\cdot}$, and row elimination transforms $\Theta_{x\cdot}$ and $w(x)$ into a macro-action as described above. If $\Theta_{x,\text{goal}} = 1$, then we fully know the state's value; this will always happen for the $|S|$th state, but may also happen earlier. We do immediate backsubstitution when this occurs, which eliminates some non-zeros above the diagonal and possibly causes other states' values to become known. Immediate backsubstitution ensures that $V(x)$ and $p_{\text{goal}}(x)$ are updated with the latest information, improving our priority estimates for states on the queue and possibly saving us work later (for example, in the case when our transition matrix is block lower triangular, we automatically discover that we only need to factor the blocks on the diagonal). Finally, all predecessors of the state popped and any states whose values became known are updated using the **update**() routine for PPI (in Figure 5).

Since S can be large, Θ will usually need to be represented sparsely. Assuming Θ is stored sparsely, GDE reduces to Dijkstra's algorithm in the deterministic case; it is easy to verify the additional matrix updates require only $O(S)$ work. In a general MDP, initially it takes no more memory to represent Θ than it does to store the dynamics of the MDP, but the elimination steps can introduce many additional non-zeros.

[6]This sequence is called the Doolittle ordering when used to compute a LU factorization.

main()
$(\forall x)\ V(x) \leftarrow M$
$V(\text{goal}) \leftarrow 0$
while (true)
    $(\forall x)\ \pi(x) \leftarrow$ undefined
    GaussDijkstraSweep()
    if ((max L_1 bellman error) $<$ tolerance)
        declare convergence
    end if
end while

GaussDijkstraSweep()
while (not queue.empty())
    $x \leftarrow$ queue.pop()
    $\pi(x) \leftarrow \arg\min_a Q(x,a)$
    $(\forall y)\ \Theta_{xy} \leftarrow P(y \mid x, \pi(x))$
    $w(x) \leftarrow c(x, \pi(x))$
    rowEliminate(x)
    $V(x) \leftarrow (\Theta_{x\cdot}) \cdot V + w(x)$
    $F = \{x\}$
    if ($\Theta_{x,\text{goal}} = 1$)
        backsubstitute(x)
    end if
    $(\forall y \in F)$ update(y)
end while

backsubstitute(x)
for each y such that $\Theta_{yx} > 0$ do
    $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x) + \Theta_{yx}$
    $w(y) \leftarrow w(y) + \Theta_{yx} V(x)$
    $\Theta_{yx} \leftarrow 0$
    if ($p_{\text{goal}}(y) = 1$)
        backsubstitute(y)
        $F \leftarrow F \cup \{\text{y}\}$
    end if
end for

rowEliminate(x)
for y from 1 to x-1 do
    $w(x) \leftarrow w(x) + \Theta_{xy} w(y)$
    $\Theta_{x\cdot} \leftarrow \Theta_{x\cdot} + \Theta_{xy}\Theta_{y\cdot}$ (1)
    $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x) + \Theta_{xy} p_{\text{goal}}(y)$
    $\Theta_{xy} \leftarrow 0$
end for
$w(x) \leftarrow w(x)/(1 - \Theta_{xx})$
$\Theta_{x\cdot} \leftarrow \Theta_{x\cdot}/(1 - \Theta_{xx})$ (2)
$\Theta_{xx} \leftarrow 0$
$p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x)/(1 - \Theta_{xx})$

Figure 6: Gauss-Dijkstra Elimination

The number of such new non-zeros is greatly affected by the order in which the eliminations are performed. There is a vast literature on techniques for finding such orderings; a good introduction can be found in [11]. One of the main advantages of GDE seems to be that for practical problems, the prioritization criteria we present produce good elimination orders as well as effective policy improvement.

Our primary interest in GDE stems from the wide range of possibilities for enhancing its performance; even in the naive form outlined it is usually competitive with PPI. We anticipate that doing "early" backsubstitution when states' values are mostly known (high $p_{\text{goal}}(x)$) will produce even better policies and hence fewer iterations. Further, the interpretation of rows of Θ as macro-actions suggests that caching these actions may yield dramatic speed-ups when evaluating the MDP with a different goal state. The usefulness of macro-actions for this purpose was demonstrated by Dean & Lin ([8]). A convergence-checking mechanism such as those used by LRTDP and HDP [6, 5] could also be used between iterations to avoid repeating work on portions of the state space where an optimal policy and value function are already known. The key to making GDE widely applicable, however, probably lies in appropriate thresholding of values in Θ, so that transition probabilities near zero are thrown out when their contribution to the Bellman error is negligible. Our current implementation does not do this, so while its performance is good on many problems, it can perform poorly on problems that generate lots of fill-in.

5 Incremental expansions

In describing IPS, PPI, and GDE we have touched on a number of methods of updating V and Q values. In summary: Value iteration iteration repeatedly backs up states in an arbitrary order. Prioritized sweeping backs up states in an order determined by a priority queue. PPI and GDE also pop states from a priority queue, but rather than backing up the popped state, they backup up all of its predecessors. IPS pops states from a priority queue, but instead of fully backing up the predecessors of the popped state x, it only recomputes Q values for actions that might reach x.

Here we provide a more thorough accounting of the expansion mechanism used by IPS. Suppose we are given an initial upper bound V_{old} on V^*. Then, we can define Q by

$$Q(x, a) = c(x, a) + \sum_y P(y \mid x, a) V_{\text{old}}(y)$$

and then V_{new} by $V_{\text{new}}(x) = \min_a Q(x, a)$. Note that rather than storing V_{new} we can simply store Q and $\pi(x)$, the greedy policy with respect to V_{old}. Our goal in an expansion operation is to set $V_{\text{old}}(x) \leftarrow V_{\text{new}}(x)$, and then update Q so it reflects this change, and then update V_{new} so that again $V_{\text{new}}(x) = \min_a Q(x, a)$. Perhaps the easiest way to ensure this property is via a *full expansion* of the state x:

```
V_old(x) ← Q(x, π(x))
for all (y, b) ∈ pred(x)
  Q(y, b) ← c(y, b) + Σ_{x'∈succ(y,b)} P(x' | y, b) V_old(x')
  if (Q(y, b) < Q(y, π(y)))
    π(y) ← b
  end if
end for
```

Doing such a full expansion requires $O(B)$ work per predecessor state-action pair. We can accomplish the same task with $O(1)$ work if we assume without loss of generality $(\forall x, a)\ P(x \mid x, a) = 0$, and perform an *incremental expansion*:

```
Δ(x) ← Q(x, π(x)) − V_old(x)
for all (y, b) ∈ pred(x)
  Q(y, b) ← Q(y, b) + P(x | y, b) Δ(x)
  if (Q(y, b) < Q(y, π(y)))
    π(y) ← b
  end if
end for
V_old(x) ← Q(x, π(x))
```

However, when doing a full expansion, we have a better option for calculating $Q(y, b)$ than the one given above. We can update $Q(y, b)$ using $Q(x', \pi(x'))$ in place of $V_{\text{old}}(x')$, and this may offer a tighter upper bound because $Q(x', \pi(x')) \leq V(x')$ when we pessimistically initialize. In our experiments, this method proved superior to doing incremental expansions, and it is the method used by Improved Prioritized Sweeping (see Figure 4 for the code). However, on certain problems incremental expansions may give superior performance. IPS based on incremental expansions tends to do more updates (at lower cost) and so priority queue operations account for a larger fraction of its running times. Thus, fast approximate priority queues might offer a significant advantage to incremental IPS implementations.

One final implementation note. Our pseudocode for IPS and PPI indicates that Q values for all actions are stored. While this is necessary if incremental expansions are performed, we do full expansions so the extra storage is not required. It is sufficient to store a single value for each state, which takes the place of Q_{old} and V in the pseudocode; newly calculated $Q(y, b)$ values can be replaced by a temporary variable; the value is only relevant if it causes $V(y)$ to change, in which case we immediately assign $V(y)$ the value of the temporary for $Q(y, b)$ rather than waiting until y is popped from the queue.

6 Experiments

We implemented IPS, PPI, and GDE and compared them to VI, Prioritized Sweeping, and LRTDP. All algorithms were implemented in Java 1.5.0 and tested on a 3Ghz Intel machine with 2GB of main memory under Linux.

Our PPI implementation uses a stabilized biconjugate gradient solver with an incomplete LU preconditioners as implemented in the Matrix Toolkit for Java [15]. No native or optimized code was used; using architecture-tuned implementations of the underlying linear algebraic routines could give a significant speedup.

For LRTDP we specified a few reasonable start states for each problem. Typically LRTDP converged after labeling only a small fraction of the the state space as solved, up to about 25% on some problems.

6.1 Experimental Domain

We describe experiments in a discrete 4-dimensional planning problem that captures many important issues in mobile robot path planning. Our domain generalizes the racetrack domain described previously in [3, 6, 5, 14]. A state in this problem is described by a 4-tuple, $s = (x, y, dx, dy)$, where (x, y) gives the location in a 2D occupancy map, and (dx, dy) gives the robot's current velocity in each dimension. On each time step, the agent selects an acceleration $a = (ax, ay) \in \{-1, 0, 1\}^2$ and hopes to transition to state $(x + dx, y + dy, dx + ax, dy + ay)$. However, noise and obstacles can affect the actual result state. If the line from (x, y) to $(x + dx, y + dy)$ in the occupancy grid crosses an occupied cell, then the robot "crashes," moving to the cell just prior to the obstacle and losing all velocity. (The robot does not reset to the start state as in some racetrack models.) Additionally, the robot may be affected by several types of noise:

- **Action Failure** With probability f_p, the requested acceleration fails and the next state is $(x + dx, y + dy, dx, dy)$.
- **Local Noise** To model the fact that some parts of the world are more stochastic than others, we mark certain cells in the occupancy grid as "noisy," along with a designated direction. When the robot crosses such a cell, it has a probability f_ℓ of experiencing an acceleration of magnitude 1 or 2 in the designated direction.

	$\|S\|$	f_p	f_ℓ	% determ	O	notes
A	59,780	0.00	0.00	100.0%	1.00	determ
B	96,736	0.05	0.10	17.2%	2.17	$\|A\| = 1$
C	11,932	0.20	0.00	25.1%	4.10	$f_h = 0.05$
D	10,072	0.10	0.25	39.0%	2.15	cycle
E	96,736	0.00	0.20	90.8%	2.41	
F	21,559	0.20	0.00	34.5%	2.00	`large-b`
G	27,482	0.10	0.00	90.4%	3.00	

Figure 7: Test problems sizes and parameters.

- **One-way passages** Cells marked as "one-way" have a specified direction (north, south, east, or west), and can only be crossed if the agent is moving in the indicated direction. Any non-zero velocity in another direction results in a crash, leaving the agent in the one-way state with zero velocity.

- **High-velocity noise** If the robot's velocity surpasses an L_2 threshold, it incurs a random acceleration on each time step with probability f_h. This acceleration is chosen uniformly from $\{-1, 0, 1\}^2$, excluding the $(0, 0)$ acceleration.

These additions to the domains allow us to capture a wider variety of planning problems. In particular, kinodynamic path planning for mobile robots generally has more noise (more possible outcomes of a given action as well as higher probability of departure from the nominal command) than the original racetrack domain allows. Action failure and high-velocity noise can be caused by wheels slipping, delays in the control loop, bumpy terrain, and so on. One-way passages can be used to model low curbs or other map features that can be passed in only one direction by a wheeled robot. And, local noise can model a robot driving across sloped terrain: downhill accelerations are easier than uphill ones.

Figure 7 summarizes the parameters of the test problems we used. The "% determ" column indicates the percentage of (s, a) pairs with deterministic outcomes.[7] The O column gives the average number of outcomes for non-deterministic transitions. All problems have 9 actions except for (B), which is a policy evaluation problem. Problem (C) has high velocity noise, with a threshold of $\sqrt{2} + \epsilon$. Figure 8 shows the 2D world maps for most of the problems.

To construct larger problems for some of our experiments, we consider linking copies of an MDP in series by making the goal state of the ith copy transitions to the start state of the $(i + 1)$st copy. We indicate k serial copies of an MDP M by M^k, so for example 22 copies of problem (G) is denoted (G^{22}).

6.2 Experimental Results

Effects of Local Noise First, we considered the effect of increasing the randomness f_ℓ and f_p for the fixed map (G), a smaller version of (B). One-way passages give this complex map the possibility for cycles. Figure 9 shows the run times (y-axis) of several algorithms plotted against f_p. The parameter f_ℓ was set to $0.5 f_p$ for each trial.

These results demonstrate the catastrophic effect increased noise can have on the performance of VI. For low-noise problems, VI converges reasonably quickly, but as noise is increased the expected length of trajectories to the goal grows, and VI's performance degrades accordingly. IPS performs somewhat better

[7] Our implementation uses a deterministic transition to apply the collision cost, so all problems have some deterministic transitions.

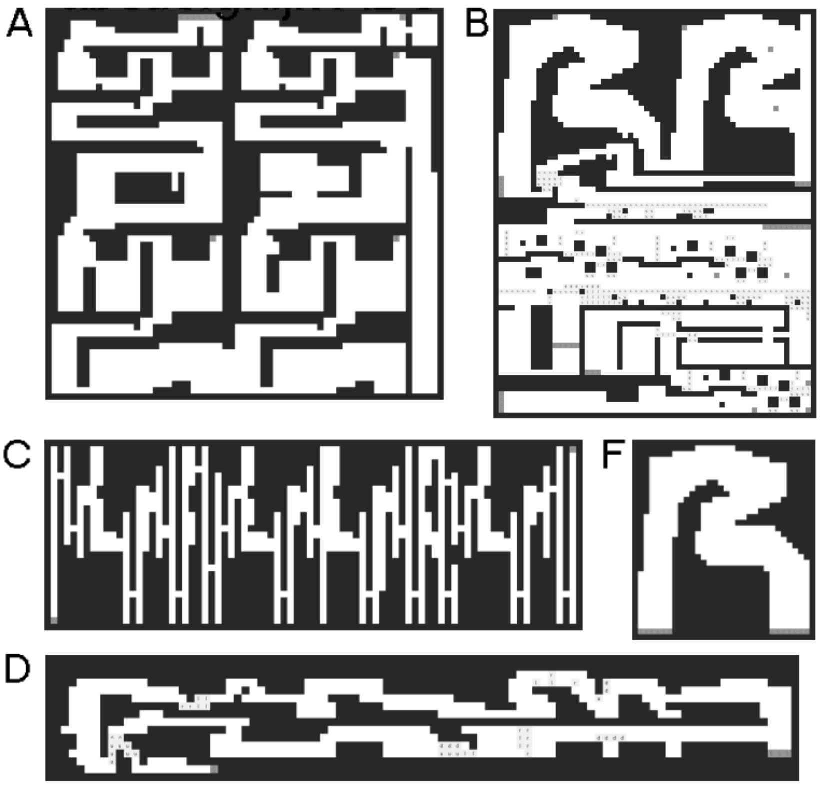

Figure 8: Some maps used for test experiments; maps are not drawn to the same scale. Problem (E) uses the same map as (B). Problem (G) uses a smaller version of map (B). Special states (one-way passages, local noise) are indicated by light grey symbols; contact the authors for full map specifications.

overall, but it suffers from this same problem as the noise increases. However, PPI's use of policy evaluation steps quickly propagates values through these cycles, and so its performance is almost totally unaffected by the additional noise. PPI-4 beats VI on all trials. It wins by a factor of 2.4 with $f_p = 0.05$, and with $f_p = 0.4$ PPI-4 is 29 times faster than VI.

The dip in runtimes for LRTDP is probably due to changes in the optimal policy, and the number and order in which states are converged. Confidence intervals are given for LRTDP only, as it is a randomized algorithm. The deterministic algorithms were run multiple times, and deviations in runtimes were negligible.

Number of Policy Evaluation Steps Policy iteration is an attractive algorithm for MDPs where policy evaluation via backups or expansions is likely to be slow. It is well known that policy iteration typically converges in few iterations. However, Figure 10 shows that our algorithms can greatly reduce the number of iterations required. In problems where policy evaluation is expensive, this can provide a significant overall savings in computation time.

The number of iterations that standard policy iteration takes to converge depends on the initial policy. We experimented with initializing to the uniform stochastic policy,[8] random policies that at least give all

[8]This is a poor initialization not only because it is an ill-advised policy, but also because it often produces a poorly-conditioned

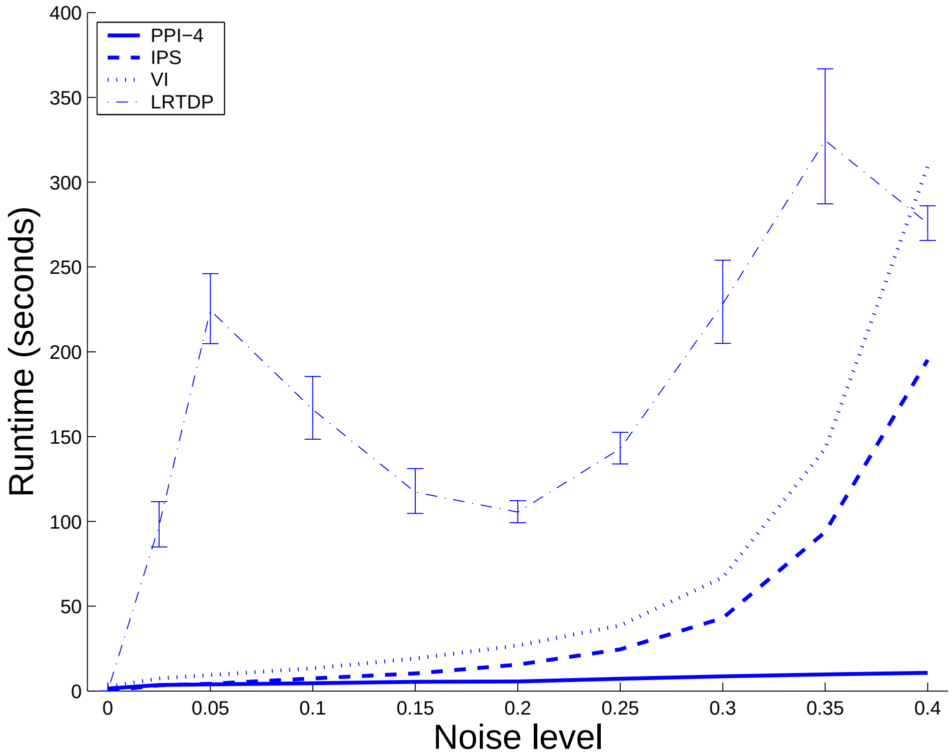

Figure 9: Effect of local noise on solution time. The leftmost data point is for the deterministic problem. Note that PPI-4 exhibits almost constant runtime even as noise is increased.

states finite value, and the optimal policy for the deterministic relaxation of the problem.[9] The choice of initial policy rarely changed the number of iterations by more than 2 or 3 iterations, and in almost all cases initializing with the policy from the deterministic relaxation gave the best performance. Policy iteration was initialized in this way for the results in Figure 10.

We compare policy iteration to PPI, where we use either 1,2, or 4 sweeps of Dijkstra policy improvement between iterations. We also ran GDE on these problems. Typically it required the same number of iterations as PPI, but we hope to improve upon this performance in future work.

Q-value Computations Our implementation are optimized not for speed but for ease of use, instrumentation, and modification. We expect our algorithms to benefit much more from tuning than value iteration. To show this potential, we compare IPS, PS, and VI on the number of Q-value computations (Q-comps) they perform. A single Q-comp means iterating over all the outcomes for a given (s, a) pair to calculate the current Q value. A backup takes $|A|$ Q-comps, for example. We do not compare PPI-4, GDE, and LRTDP based on this measure, as they also perform other types of computation.

IPS typically needed substantially fewer Q-comps than VI. On the deterministic problem (A), VI required 255 times as many Q-comps as IPS, due to IPS's reduction to Dijkstra's algorithm; VI made 7.3

linear system that is difficult to solve

[9]This is the policy chosen by an agent who can choose the outcome of each action, rather than having an outcome sampled from the problem dynamics. The value function for this policy can be computed by any shortest path algorithm or A^* if a heuristic is available.

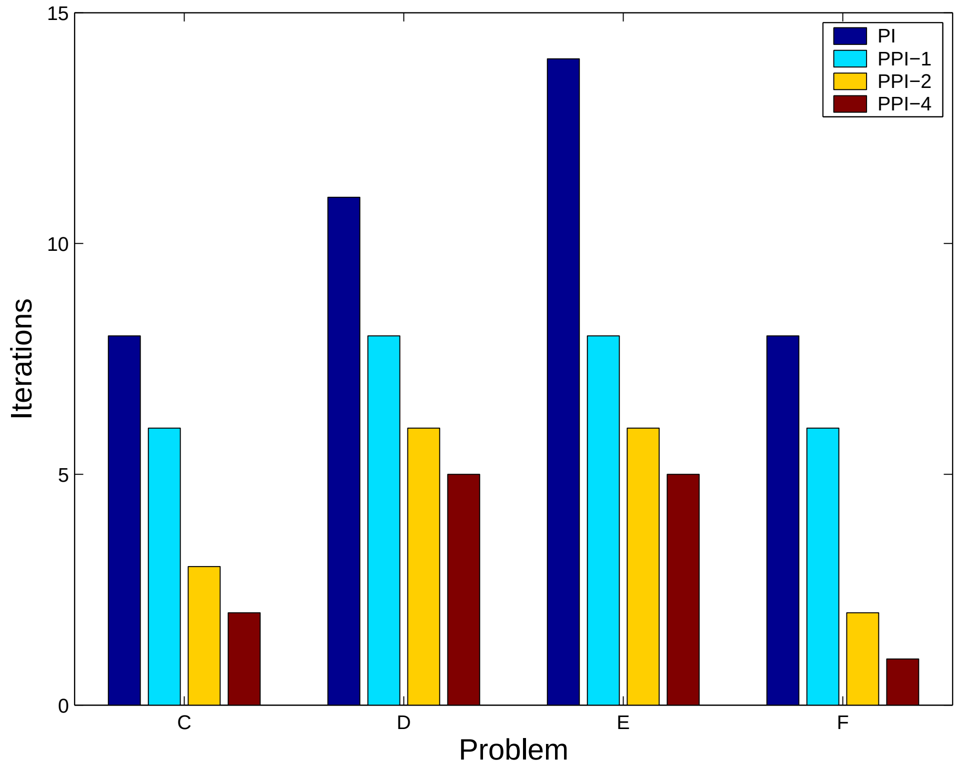

Figure 10: Number of policy evaluation steps.

times as many Q-comps as PS. On problems (B) through (F), VI on average needed 15.29 times as many Q-comps as IPS, and 5.16 times as many as PS. On (G^{22}) it needed 36 times as many Q-comps as IPS. However, these large wins in number of Q-comps are offset by value iteration's higher throughput: for example, on problems (B) through (F) VI averaged 27,630 Q-comps per millisecond, while PS averaged 4,033 and IPS averaged 3,393. PS and IPS will always have somewhat more overhead per Q-comp than VI. However, replacing the standard binary heap we implemented with a more sophisticated algorithm or with an approximate queuing strategy could greatly reduce this overhead, possibly leading to significantly improved performance.

Figure 11 compares the number of Q-comps required to solve serially linked copies of problem (D): the x-axis indicates the number of copies, from (D^1) to (D^8). VI still has competitive run-times because it performs Q-comps much faster. On (D^8) it averages 41,360 Q-comps per millisecond, while PS performs only 4,453 and IPS only 3,871.

Overall Performance of Solvers Figure 12 shows a comparison of the run-times of our solvers on the various test problems. Problem (G^{22}) has 623,964 states, showing that our approaches can scale to large problems.[10] On (G^{22}), the stabilized biconjugate gradient algorithm failed to converge on the initial linear systems produced by PPI-4, so we instead used PPI where 28 initial sweeps were made (so that there was a reasonable policy to be evaluated initially), and then 7 sweeps were made between subsequent evaluations.

[10] This experiment was run on a different (though similar) machine than the other experiments, a 3.4GHz Pentium under Linux with 1GB of memory.

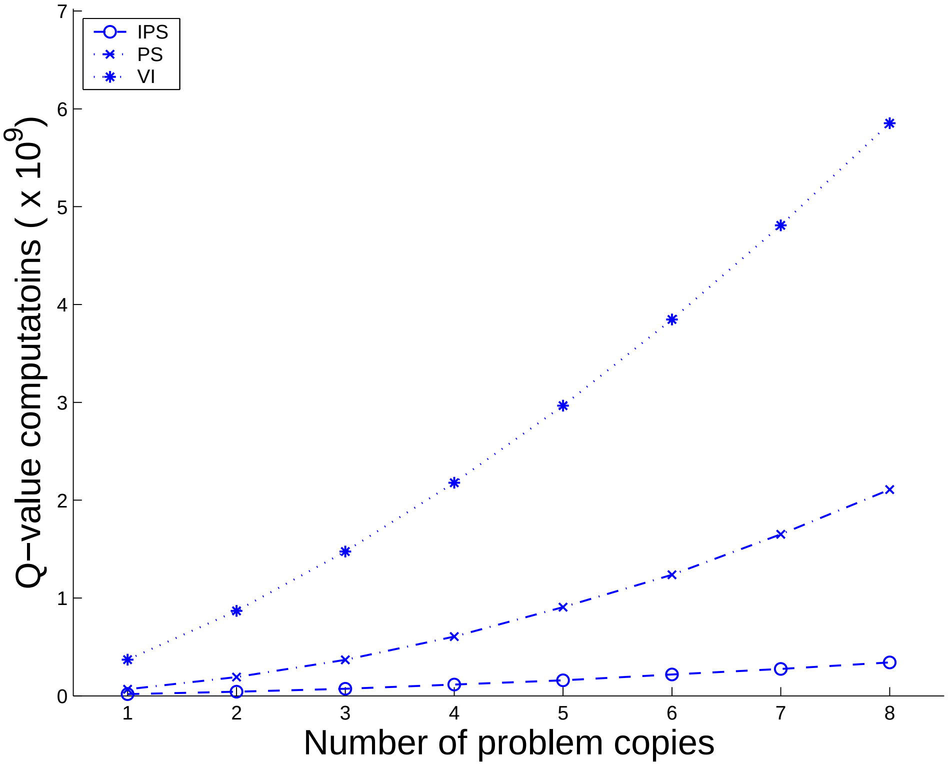

Figure 11: Comparison of number of Q-computations performed by IPS, PS, and VI to solve serially-linked copies of problem (D).

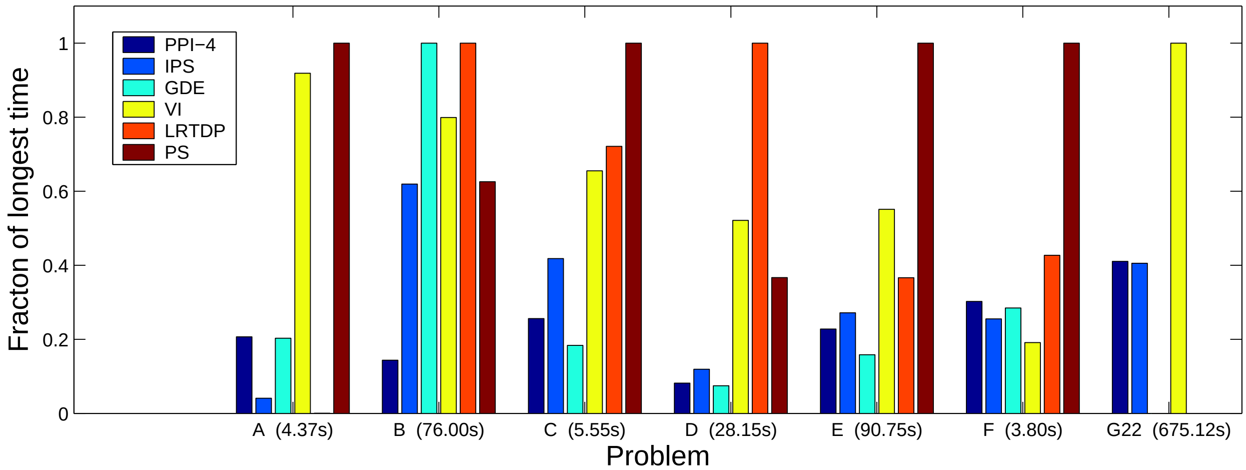

Figure 12: Comparison of a selection of algorithms on representative problems. Problem (A) is deterministic, and Problem (B) requires only policy evaluation. Results are normalized to show the fraction of the longest solution time taken by each algorithm. On problems (B) and (E), the slowest algorithms were stopped before they had converged. LRTDP is not charged for time spent calculating its heuristic, which is negligible in all problems except (A).

We also found that adding a pass of standard greedy policy improvement after the sweeps improved performance. These changes roughly balanced the time spent on sweeping and policy improvement. In future work we hope to develop more principled and automatic methods for determining how to split computation time between sweeps and policy evaluation. We did not run PS, LRTDP, or GDE on this problem.

Generally, our algorithms do best on problems that are sparsely stochastic (only have randomness at a few states) and also on domains where typical trajectories are long relative to the size of the state space. These long trajectories cause serious difficulties for methods that do not use an efficient form of policy evaluation. For similar reasons, our algorithms do better on long, narrow domains rather than wide open ones; the key factor is again the expected length of the trajectories versus the size of the state space.

Value iteration backed up states in the order in which states were indexed in the internal representation; this order was generated by a breadth-first search from the start state to find all reachable states. While this ordering provides better cache performance than a random ordering, we ran a minimal set of experiments and observed that the natural ordering performs somewhat worse (up to 20% in our limited experiments) than random orderings. Despite this, we observed better than expected performance for value iteration, especially as it compares to LRTDP and Prioritized Sweeping. For example, on the `large-b` problem (F), [5] reports a slight win for LRTDP over VI, but our experiments show VI being faster.

Also, GDE's performance is typically close to or better than that of PPI-4, except on problem (B), where GDE fails due to moderately high fill in. These results are encouraging because GDE already sometimes performs better than PPI-4, and currently GDE is based on a naive implementation of Gaussian elimination and sparse matrix code. The literature in the numerical analysis community shows that more advanced techniques can yield dramatic speedups (see, for example, [13]), and we hope to take advantage of this in future versions of GDE.

7 Discussion

The success of Dijkstra's algorithm has inspired many algorithms for MDP planning to use a priority queue to try to schedule when to visit each state. However, none of these algorithms reduce to Dijkstra's algorithm if the input happens to be deterministic. And, more importantly, they are not robust to the presence of noise and cycles in the MDP. For MDPs with significant randomness and cycles, no algorithm based on backups or expansions can hope to remain efficient. Instead, we turn to algorithms which explicitly solve systems of linear equations to evaluate policies or pieces of policies.

We have introduced a family of algorithms—Improved Prioritized Sweeping, Prioritized Policy Iteration, and Gauss-Dijkstra Elimination—which retain some of the best features of Dijkstra's algorithm while integrating varying amounts of policy evaluation. We have evaluated these algorithms in a series of experiments, comparing them to other well-known MDP planning algorithms on a variety of MDPs. Our experiments show that the new algorithms can be robust to noise and cycles, and that they are able to solve many types of problems more efficiently than previous algorithms could.

For problems which are fairly close to deterministic or with only moderate noise and cycles, we recommend Improved Prioritized Sweeping. For problems with fast mixing times or short average path lengths, value iteration is hard to beat and is probably the simplest of all of the algorithms to implement. For general use, we recommend the Prioritized Policy Iteration algorithm. It is simple to implement, and can take advantage of fast, vendor-supplied linear algebra routines to speed policy evaluation.

8 Acknowledgements

The authors wish to thank the reviewers of [16] for helpful comments, and Avrim Blum and Maxim Likhachev for useful input. This work was funded in part by DARPA's MICA project, AFRL contract F30602-01-C-0219.

References

[1] D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1998.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, 1995.

[4] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Massachusetts, 1995.

[5] Blai Bonet and Hector Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In G. Gottlob, editor, *Proc. 18th International Joint Conf. on Artificial Intelligence*, pages 1233–1238, Acapulco, Mexico, 2003. Morgan Kaufmann.

[6] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. of ICAPS-03*, pages 12–21, 2003.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[8] T. Dean and S. Lin. Decomposition techniques for planning in stochastic domains. In *IJCAI*, 1995.

[9] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artif. Intell.*, 76(1-2):35–74, 1995.

[10] T. G. Dietterich and N. S. Flann. Explanation-based learning and reinforcement learning: A unified view. In *12th International Conference on Machine Learning (ICML)*, pages 176–184. Morgan Kaufmann, 1995.

[11] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Oxford, 1986.

[12] David Ferguson and Anthony (Tony) Stentz. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[13] Anshul Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Softw.*, 28(3):301–324, 2002.

[14] Eric A. Hansen and Shlomo Zilberstein. LAO*: a heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, 129(1-2):35–62, 2001.

[15] Bjrn-Ove Heimsund. Matrix toolkits for java (MTJ). `http://www.math.uib.no/~bjornoh/mtj/`, 2004.

[16] H. Brendan McMahan and Geoffrey J. Gordon. Fast exact planning in markov decision processes. In *To appear in ICAPS*, 2005.

[17] Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[19] Marco Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam, 1999.