Focussed Processing of MDPs for Path Planning

Dave Ferguson and Anthony Stentz
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{dif, tony}@cmu.edu

Abstract

We present a heuristic-based algorithm for solving restricted Markov decision processes (MDPs). Our approach, which combines ideas from deterministic search and recent dynamic programming methods, focusses computation towards promising areas of the state space. It is thus able to significantly reduce the amount of processing required to produce a solution. We demonstrate this improvement by comparing the performance of our approach to the performance of several existing algorithms on a robotic path planning domain.

1. Introduction

Markov decision processes (MDPs) have been widely used as models for uncertainty-based reasoning in AI due to their generality and intuitive appeal. However, classical methods for solving MDPs require time at best polynomial in the number of states in the domain [7]. When dealing with large state spaces, this can become extremely cumbersome. A number of researchers have investigated ways to reduce this computation by restricting the number of states evaluated or processing the states in a particular order.

In this paper, we present a heuristic algorithm for solving MDPs that borrows ideas from these recent MDP algorithms as well as from classical deterministic planning. Our approach focusses attention on areas of the state space that appear most promising and processes these areas in an order that reduces the overall computation required.

We begin by reviewing MDPs and classical techniques for solving them. In Section 3, we describe a number of more recent approaches that attempt to exploit different characteristics of the problem in order to reduce the amount of processing performed. In Section 4, we present our algorithm and explain some of the intuition behind it. We then provide comparative results for all the algorithms on a robotic path planning domain and conclude in Section 6 with additional discussion.

2. Markov Decision Processes

A Markov decision process is defined as a tuple (S, A, P, C), where S is the set of states of the world (some of which may be terminal), A is the set of actions available in every state, $P : S \times A \times S \rightarrow [0, 1]$ is the transition model such that $P(s_i, a, s_j)$ is the probability of transitioning to state s_j when performing action a in state s_i, and $C : S \times A$ is the cost function, where $C(s_i, a)$ specifies the expected cost of taking action a in state s_i. A policy $\pi : S \rightarrow A$ is a mapping from states to actions, specifying an action to be taken in each world state.

Given a policy π and a cost function C, we can define the value of a state s_i to be its expected total (undiscounted) cost under the policy π. Given this definition, the value of state s_i given policy π can be written as [11]:

$$V_\pi(s_i) = C(s_i, \pi(s_i)) + \sum_{s_j \in S} P(s_i, \pi(s_i), s_j) \cdot V_\pi(s_j).$$

A policy π is said to be optimal if $V_\pi(s_i) \leq V_{\pi'}(s_i)$ for all $s_i \in S$ and policies π'. Given an optimal policy π, V_π is known as the optimal value function. Typically, we are trying to calculate these two elements.

2.1. Path Planning MDPs

In outdoor mobile robot navigation, GPS is used to provide accurate global position estimation. However, in forest areas or valleys, GPS coverage is scarce. When this is the case, a robot has to rely on relative position estimation (known as dead-reckoning) and error can accrue. This means that it may be possible for a robot to deviate from its intended course when it is traversing across areas without GPS coverage. If the robot knows apriori which areas have GPS coverage and which do not, it can incorporate this uncertainty into the planning task.

We can do this with an MDP, where the states of the MDP represent areas of the environment with GPS coverage and the actions provide nondeterministic transitions between these GPS areas. The cost of a state-action pair $C(s_i, a)$ is based on the difficulty of the terrain associated with state s_i and all adjacent states possibly traversed during the course of applying action a.

There are a few key properties of this path planning domain which set it apart from the general MDP framework.

Firstly, it has a *low dispersion rate* [7]: from any state in the environment, there are only a few neighboring states into which an agent may move directly. Secondly, there are specified *start* and *goal* states: an agent is trying to determine a path from the start state to the goal state. Finally, the uncertainty associated with actions is limited: given realistic dead-reckoning error (see [15]), the number of possible states an agent may end up in after applying an action in a given state is very small. We will describe the models we use for experimentation in more depth in Sections 4 and 5.

As we will see, these properties enable one to use clever algorithms to reduce the amount of computation necessary to solve path planning MDPs. These algorithms are discussed in Sections 3 and 4.

2.2. Classical Approaches

An optimal policy and value function for an MDP can be computed using the classical approaches of value iteration [2], policy iteration [11], or linear programming [3, 16]. We restrict our attention here to the former two methods.

Value iteration works by treating the value equation as an assignment. It starts with an initial upper bound value $V^{(0)}$ and at iteration i it sets

$$V^{(i+1)}(s_i) = \min_{a \in A} \sum_{s_j \in S} P(s_i, a, s_j) \cdot (C(s_i, s_j) + V^{(i)}(s_j))$$

for each element $s_i \in S$. It provably converges to the optimal value function [5] and the optimal policy can be extracted by picking the action in each state which achieves the minimum value specified by the value function.

Policy iteration works by maintaining a current policy $\pi^{(i)}$ at each step i. It solves for the value function of this policy using Equation (1), then allows the policy to be updated so that the action taken in each state provides the minimum value relative to the newly computed values for the state's neighbors. This process repeats until the policy does not change between iterations, i.e., $\pi^{(i+1)} = \pi^{(i)}$. This algorithm also provably converges.

Both of these approaches require a number of value updates (the number of times the value equation must be performed) polynomial in the number of states in the world [4]. Typically, value iteration is more efficient when the number of applicable actions in each state is small.

3. Related Approaches

Both value iteration and policy iteration are simple algorithms guaranteed to produce optimal results. But often the computation required to generate these results is far more than is necessary. In particular, when we are only interested in the values and policies of a restricted subset of the states in the world, it may be more sensible to restrict our attention to these states. Furthermore, in domains such as robotic path planning, it can make a huge difference if we order our value updates in a sensible manner.

The following approaches attempt to gain computational advantage over the classical methods by paying close attention to the nature of the particular problem being solved. As a result, they are often able to produce solutions much more efficiently. We present each as it applies to the path planning MDP framework described above.

3.1. Real-Time Dynamic Programming

In [1], Barto et al. introduce the Real-Time Dynamic Programming (RTDP) algorithm. The algorithm attempts to restrict the number of examined states to a small fraction of the complete state space.

The algorithm begins with admissible value estimates for each state in the world. It then performs a series of simulated traverses through the environment that begin at the start state and follow the greedy policy with respect to the current value estimates. The values of states are updated using the value equation as they are encountered along these traverses.

The algorithm has been shown to converge to the optimal value function on the set of all states reachable from the initial state under the optimal policy [1]. It has also been extended by Bonet and Geffner [6] to Labeled RTDP (LRTDP), which helps speed up the convergence of RTDP by labelling as solved states that have fully converged so that subsequent traverses need not venture past these states.

3.2. Envelope Propagation

Dean et al. [7] describe a related method of reducing the state space to an "envelope" of consideration. As with RTDP, the Envelope Propagation (EP) algorithm begins with admissible value estimates for each state in the world. First, they generate ten depth-first paths from the start state to the goal state. They then remove redundant steps in each of these paths and select the shortest. Given this path as their initial envelope, they "strengthen" it by adding neighboring cells which may help increase the chance of an agent staying within the envelope if it tries to follow the policy induced by the path.

They then alternate between two phases: envelope expansion and policy generation. The envelope expansion phase consists of simulating a number of runs through the envelope from the start state, given the current policy. If a run encounters a state that is not in the envelope, then that state is marked as a potential element to be added to the envelope. After all the simulated runs are performed, the states that were encountered most often are added to the envelope. The envelope is then strengthened by computing paths from each of these new states back into the envelope and adding the states along these paths to the envelope also.

The policy generation phase consists of performing dynamic programming over the envelope to compute an optimal (relative to the envelope) policy for each state in the envelope. This can be performed using either policy iteration or value iteration.

3.3. LAO*

Similar in practise to the envelope propagation approach of Dean et al. is the LAO* algorithm [10, 8]. LAO* also alternates between an expansion phase and a policy generation phase. However, its expansion phase is slightly different from that of envelope propagation. A "fringe" of states is maintained that represents all states adjacent to the current envelope that are reachable from the start state given the current policy. During expansion, the entire fringe is added to the envelope. Thus, LAO* can increase its envelope quite substantially at each expansion phase.

It is worth noting that the motivation behind LAO* was the extension of the classic search algorithm AO* to handle cyclic domains such as MDPs [10]. The application of deterministic heuristic search techniques on stochastic domains has proven to be very fruitful. The research we present here is likewise an attempt to capture the intuition behind a number of classical search ideas and employ them in the development of effective stochastic algorithms.

3.4. Prioritized Sweeping

The Prioritized Sweeping (PS) algorithm was developed by Moore and Atkeson to perform reinforcement learning on stochastic Markov systems [13]. Rather than performing a number of passes through the environment in which every state value is updated (as value iteration does), Prioritized Sweeping maintains a priority queue and updates states of the world based on their priority in this queue. There are two possible ways of employing PS on the path planning MDP. The first is to initialise all states with admissible heuristic values and then propagate cost updates outwards from the start. The second is to initialise all states (except for the goal) with infinite values and propagate cost updates from the goal inwards. In the former case, the priority queue is seeded with the start state, while in the latter case, it is seeded with the states neighboring the goal. In our experiments we found the latter approach more effective.

In both cases, the algorithm then repeats the following steps. The state s with maximum priority is popped off the queue and a value update is performed for s. The difference in the value of s before and after the update is recorded as its *delta value*, Δ. Each state that neighbors s is then placed onto the queue with priority Δ (or has its priority updated if it is already on the queue with a lower priority). This process continues until the priority queue is empty or an acceptable solution has been reached.

Prioritized Sweeping focusses its computation on areas of the environment that are experiencing the greatest change in value during propagation. Often, these areas are the most interesting, and by updating them first, their resulting values can be used to more accurately update subsequent states.

```
While termination criteria not satisfied

1. Pop the state with minimum key value from the
   queue. Call this state $x$.

2. For each state $s \in \{x \cup pred(x)\}$
2.2 $V'(s) := \min_{a \in A} \sum_{s_j \in S} P(s, a, s_j) \cdot (C(s, s_j) + V(s_j))$
2.3 $\Delta := |V(s) - V'(s)|$
2.4 $V(s) := V'(s)$
2.5 If $\Delta > \epsilon$
      (i) $\mathcal{H}(r, s) :=$ Heuristic Cost from start state to $s$.
     (ii) $\mathcal{G}(s, g) :=$ Heuristic Cost from $s$ to goal.
    (iii) $\mathcal{K} := \mathcal{H}(r, s) + \mathcal{G}(s, g)$
     (iv) Insert $s$ onto queue with key value $\mathcal{K}$.
```

Figure 1. Focussed Dynamic Programming

4. Focussed Dynamic Programming

Each of the above approaches uses one of two methods to reduce the amount of computation required. The first method, used by RTDP, LRTDP, EP, and LAO*, is to restrict the set of considered states to include only those that are completely necessary for obtaining an optimal solution. This allows these approaches to ignore potentially large sections of the world and thus reduce the number of value updates required.

The second method, used by PS, is to pay particular attention to the order in which states are updated, so that each update can be as effective as possible. By concentrating on areas of the world which are experiencing the greatest change in value, PS is able to direct its value propagation from areas that have had their values updated towards areas that have not.

Our new algorithm, *Focussed Dynamic Programming* (FP), attempts to capture the benefits of both of these methods. It uses heuristics to limit the number of states examined and focusses value updates so that they are used most effectively.

Like backwards A* [14], our algorithm propagates out from the goal state and focusses towards the start state. It selects states to update based on a heuristic estimate of their value and a heuristic cost to the start state. Both of these considerations are vital: incorporating the heuristic estimate of a state's value helps ensure that states that could have low optimal values but that have high current values are updated, and incorporating the heuristic cost to the start state favors states that are likely to have the greatest influence on the value of the start state.

All states are initially assigned infinite values, and the value of a state at any time is an upper bound of the state's optimal value. The algorithm maintains a priority queue of states to be updated, ordered by increasing *key values*. The key value of a particular state s is the heuristic cost from the start state to s plus the (continuously updated) heuristic cost from s to the goal. These values are similar to the f values used in A*; however, they differ in that the current cost of the state is not incorporated into the key value. The queue is initialized with the goal state, whose key value is just its heuristic cost from the start state.

When a state s is popped off the priority queue, s and all of its neighboring states have their values updated. As with PS, each of these states computes its own Δ: the difference in its value before and after the update. If the Δ for a particular state is greater than some tiny threshold, the state is added back onto the queue with its new key value. If the state is already on the queue, it is promoted if its new key value is less than its previous one. In this way, the Δ values are used to decide when to insert a state onto the queue, but they do not have any influence on the state's priority within the queue.

The main loop of the algorithm is given in Figure 1. Here, $pred(x)$ refers to all predecessor states of x, i.e., all states s such that there exists some action a that transitions s to x with some non-zero probability, $V(s)$ stores the current value of state s, and ε is the threshold value mentioned earlier.

Any admissible heuristic function can be used for $\mathcal{H}(r,s)$, but ideally it will satisfy $\mathcal{H}(s,s) = 0$ and $\mathcal{H}(s,x) \leq \mathcal{H}(s,s') + c^*(s',x)$, where $c^*(s',x)$ is the minimum possible path cost from state s' to state x.

The calculation of the heuristic value of a state, $\mathcal{G}(s,g)$, is a little more complicated. In deterministic search algorithms such as A* this would amount to the current value of s: the value of the state which placed s on the queue plus the cost of transitioning to that state from s. But in non-deterministic domains, the current value of s may depend on *several* states. Some of these states may not have been updated yet, leaving them with unrealistically large values. We would like the priority of a state on the queue to represent the current promise of the state, i.e., an indication of what the value of the state *could* be based on the values of states which have already converged. This would allow us to determine which states' values were worth spending more time converging, and which could be ignored as irrelevant.

To do this, we set $\mathcal{G}(s,g)$ to be a *lower bound for the value of s given the current values of converged states*. In the present domain, this is performed by computing a heuristic value for each possible action a from state s. For each of the possible resulting states after performing action a (with intended outcome state s_d) we use one of two values. If the state is an obstacle, we use a prohibitively large value. If the state is not an obstacle, we use the value of state s_d. The heuristic value associated with action a is then given by

$$\mathcal{G}_a(s,g) = \sum_{s_j \in S} P(s_i, a, s_j) \cdot (C(s_i, s_j) + V_{(a,s)}(s_j)),$$

where $V_{(a,s)}(s_j)$ is the value used for state s_j when looking at action a in state s, as just described. $\mathcal{G}(s,g)$ is then taken to be the minimum heuristic value associated with any action from s:

$$\mathcal{G}(s,g) = \min_{a \in A} \mathcal{G}_a(s,g).$$

Finally, if the resulting value of $\mathcal{G}(s,g)$ is greater than the current value of s, we set $\mathcal{G}(s,g) = V(s)$ to ensure that $\mathcal{G}(s,g)$ is a guaranteed lower bound.

The intuition behind our heuristic function $\mathcal{G}(s,g)$ is as follows. State s had to be inserted on the queue by one of its neighbors, which at the time had the lowest key value of all states on the priority queue. If the heuristic value of s is computed as above, then s can at best have a key value equal to the state which inserted it onto the queue. If this key value is small enough for s to be the most promising state (i.e., at the top of the priority queue), then the state which inserted s onto the queue must have converged to its optimal value given the values of states within the current "envelope": the states which have already been popped off the priority queue. From this point, s will keep being popped

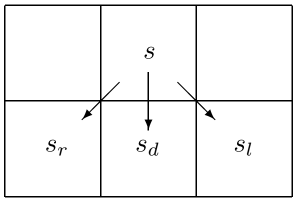

Figure 2. The possible resulting states after attempting to move from state s to state s_d using our action model.

until either it has converged (at which point its Δ will be zero and it will not get reinserted onto the queue) or some other state appears more promising.

The resulting algorithm captures the benefits of A* while contending with the complex interrelationship of state values inherent in nondeterministic domains such as MDPs. Note that states which have converged relative to the current envelope do not necessarily have optimal values - it is possible for new states to be popped which may eventually reduce the values of states already converged. This is akin to envelope expansion in [7] and [10].

The termination criteria we use is that of backwards A*: when the lowest key value on the priority queue is greater than the value of the start state. This criteria does not guarantee that the value of the start state will be completely optimal, as we will discuss, but in practise we have found it to generate very accurate results.

5. Experiments and Results

To test both the algorithm and the termination criteria independently, we performed two different experiments. For use in each experiment we constructed a set of uniform 8-connected grid environments, each of dimension 200×200. We varied the number of untraversable (or obstacle) regions in the environments so that we had 20 maps for each obstacle density from 0 (no untraversable areas) to 20 (one region out of every five is untraversable). In each map, the cost of traversing between two adjacent non-obstacle grid areas was randomly generated. The start and goal states were the same for each environment: the start was the grid cell at the center of the left edge of the environment with the goal at the center of the right edge. We used Euclidean distance for our heuristic $\mathcal{H}$.

To model the uncertainty associated with dead-reckoning, we employed a transition function that assigned non-zero probability to three adjacent states for each action. We have did not allow for the possibility that an action could take the agent in an opposite direction from that intended; dead-reckoning error is not nearly this extreme over reasonable distances. However, in our discussion we do mention generalising the transition model (from having three possible resulting states to five) to show that the relative efficiency of our algorithm is not completely tied to our current action model. Of course, as with all exploitative MDP approaches, as the number of consequences of each action grows, the number of relevant states and computation time required to reach a solution increase significantly.

Clearly, by using a grid-based representation with stochasticity modelled as discrete transitions, we are only providing an approximation of the true behavior of the agent. However, as shown in [12], incorporating stochasticity of this nature can still provide very robust trajectories.

Figure 2 shows the transition function used. We set $P(s, a, s_d) = 0.85$ and $P(s, a, s_l) = P(s, a, s_r) = 0.075$. Using this model, Figure 3 shows the states examined leading up to various stages of the FDP algorithm when applied to a small example map. Each cell represents a GPS area. The start state is the blue (darkly shaded) cell at the far left center; the goal is the blue (darkly shaded) cell at the far right. We have lightly shaded the other cells according to the cost of traversing out of their respective areas; black cells represent untraversable areas of the environment. Note that the red (darkly shaded) cells are the only cells which would be examined regardless of the size of the map.

For our first experiment, we initially ran value iteration until it converged on an optimal value for each state in the world [1]. We then ran our Focussed Dynamic Programming (FP) approach until its termination criteria was satisfied. We recorded the error of the FP value for the start state, then ran six different algorithms until they each reached a value for the start state that was within the error achieved by FP. In order, the algorithms compared were: value iteration (VIS), EP, LAO* (LAO), RTDP (RT), LRTDP (LR), and PS. We recorded both the number of value updates and CPU time required by each approach when run on a P3 1.4 GHz processor.

In the second experiment, we altered the termination condition of the FP algorithm so that it finished as soon as the value of the start state was within some error δ of its optimal value. We then ran each of the above algorithms over this same error threshold and recorded their relative performances.

For brevity, we have not included all our experimental results, nor their statistical measures of accuracy. The comprehensive set can be found in [9].

1 The absolute optimal values would require us running value iteration indefinitely. We chose to terminate when the maximum Δ value over all states was less than 10^{-3}. Because we are dealing with an undiscounted MDP, we cannot easily guarantee bounds on the error of the values achieved using such a termination condition, but in practise we found this to be a rigorous enough criteria to provide convergence.

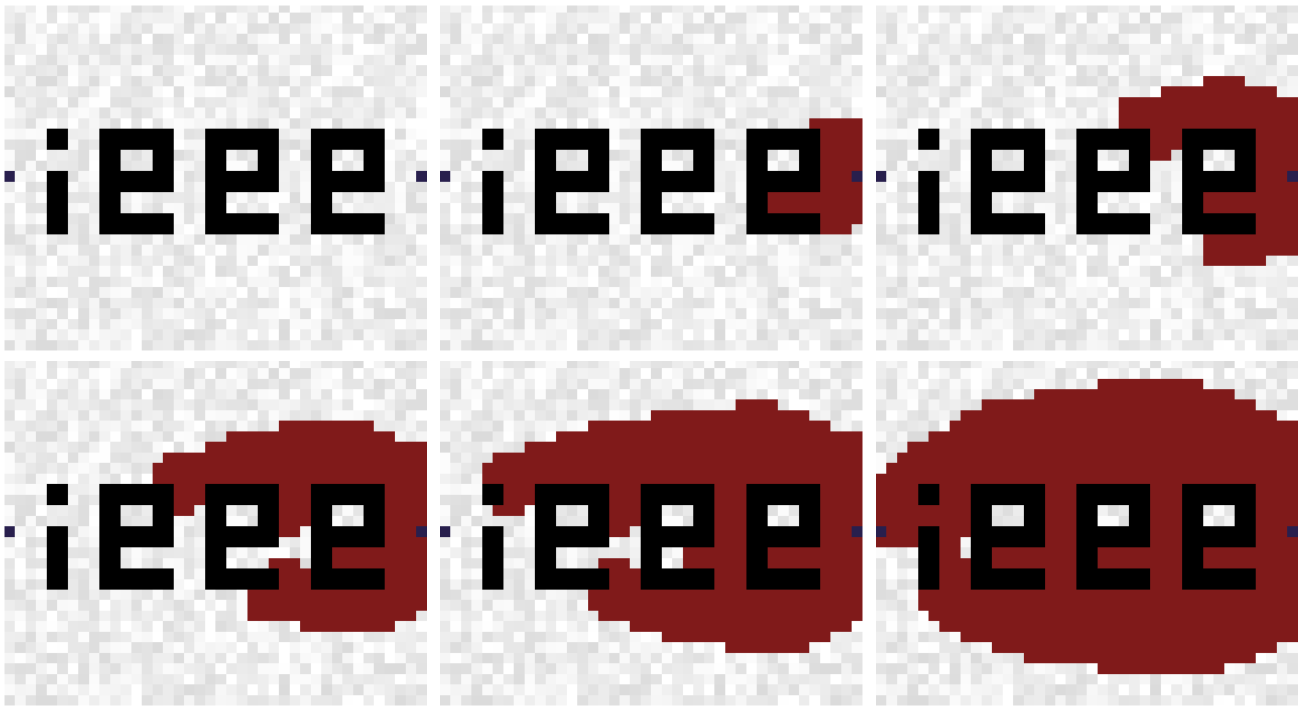

Figure 3. Example of Focussed Dynamic Programming in action. The initial state is on the far left with the goal on the far right (both darkly shaded). In red (darkly shaded) are the states examined during the course of the algorithm. The sequence runs from left to right, top to bottom.

OD	Average Number of Value Updates (times 10^6)							Average Time Taken (in seconds)							Error
	FP	VIS	EP	LAO	RT	LR	PS	FP	VIS	EP	LAO	RT	LR	PS	(%)
0	0.2	0.8	1.7	2.2	4.4	2.4	1.7	0.2	0.5	2.3	1.3	3.5	1.7	6.6	0.01
5	0.2	1.0	3.9	2.6	9.8	1.8	1.5	0.2	0.6	8.0	1.7	8.0	1.4	6.7	0.00
10	0.2	1.5	6.5	3.0	14.1	1.5	1.5	0.2	0.9	20.5	2.0	12.1	1.3	6.4	0.52
15	0.2	1.7	9.2	3.9	37.1	2.4	2.1	0.2	1.2	41.4	2.8	32.0	1.9	9.4	0.00
20	1.0	16.6	39.8	12.6	6.1	14.8	21.6	1.3	11.9	167	9.5	5.7	11.4	101	1.74

Table 1. The error associated with the value of the start state as returned by Focussed Dynamic Programming. Also shown are the number of value updates and processing time required by each competing approach to generate solutions within this error range.

The number of updates and CPU time required by each approach in the first experiment are displayed in Table 1. The leftmost column represents the obstacle density (OD) of each set of environments. The error of the FP result is shown on the rightmost column, as a percentage of the optimal cost. In both the number of value updates and the total time taken, the FP approach is significantly more efficient than any of the other algorithms.

Partial results for the second experiment are shown in Figure 4. For this experiment, we set $\delta = 0.1$ (where optimal values for the start state ranged from 250 to 900).

6. Discussion

It can be seen from the results that FP offers a significant improvement over current approaches. In this section, we discuss and explain the relative performance of each of the algorithms with reference to their unique characteristics.

VIS performs favorably because it begins with the entire state space as its "envelope", so it does not spend a lot of time processing intermediate envelopes, where states cannot approach their optimal values because of the restricted

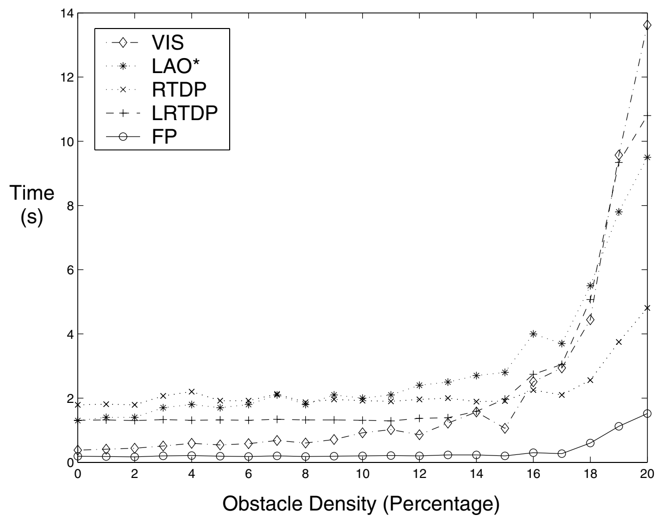

Figure 4. Run time required for the five most efficient approaches to get within 0.1 of the optimal value for the start state.

nature of the envelope. For the current experiments, the start and goal states were on opposite ends of the environment, and thus a significant proportion of the state space was relevant. If the position of the start and goal states were closer to one another relative to the size of the environment then VIS would not perform nearly as well.

EP takes much longer than VIS for two main reasons. Firstly, its use of simulated runs to expand its envelope is both time consuming and can impose limitations on how many states are added to the envelope at each expansion. As a result, EP spends a significant amount of time processing intermediate envelopes. Secondly, the use of admissible values by EP typically causes dynamic programming to take much longer to converge than if upper bounded values were employed.

LAO* shares the same disadvantage as EP of processing intermediate envelopes, but because it adds the entire fringe at each stage, it saves on the computation of simulating runs and its envelope expands at a much faster rate.

In general, RTDP performs worse than LAO* when the desired solution must be very close to optimal. Because RTDP only updates during its simulated runs, it is prone to not updating the values of states associated with highly unlikely paths. This means that occasionally it can take a very long time to get the value of the start state to be within a small error of its optimal value. In the tabulated results we can see a couple such situations. However, when our error threshold is less demanding (such as the 0.1 error bound in the second set of experiments), RTDP performs much more favorably. In this setting, the fact that RTDP does not deal with the entire envelope every time it updates values is highly beneficial, and gives it an edge over LAO*.

LRTDP is able to achieve even better performance than RTDP because it does not need to reprocess states whose values have already converged. This enables it to obtain near-optimal results (such as those in our first experiment) much more efficiently.

PS consistently requires more time than all other approaches except EP. This is because it fails to focus its propagation towards the start state. By not incorporating into the priority of a state its potential influence on the value of the start state, PS ends up performing far more value updates than are necessary.

PS was designed to be used to update universal MDP policies when new or conflicting information is received and optimality is not the primary concern. It has also proven to be very useful for learning the state transition functions and rewards of MDPs. Under these situations, its propagation based on cost differences is much more beneficial.

FP performs well because it is able to grow its envelope out from the goal, so that states which get converged early generally do not need to be updated again. Thus, it does not need to converge its entire envelope every time it is expanded. Secondly, it focusses its propagation towards the start state. The combination of these two characteristics enable it to minimise both the number of states examined and the number of updates required to converge each examined state. As a result, it is able to produce solutions much more efficiently than any of the competing approaches.

It is worth noting that the performance of FP is not intricately linked to the current domain. We ran similar experiments using an action model that assigned non-zero probabilities to transitioning to one of five neighboring states for each action and it still showed significant performance gains over the other approaches (see [9] for results). However, we are of the opinion that this added complexity is unjustified for our current robotic path planning domain.

Finally, the termination criteria used by FP in the first set of experiments has shown itself to be quite effective. All of the approaches discussed here can return optimal solutions, yet knowing when to stop processing because the current solution is “good enough” can be difficult. Continuing until the envelope is completely closed (for EP and LAO*) or until there are no more states on the queue (PS) returns solutions of far more accuracy than we typically require, yet the bounds that can be computed for intermediate solutions are often very loose [10]. We have presented a termination condition that allows for good results (on average within 0.18 percent of optimal over environments with obstacle densities of up to 20 percent) to be generated very quickly.

We have also extended FP to an unfocussed variety which can provide the same error performance as value iteration, including convergence to the optimal solution. To do this, we simply process *all* states in the environment in or-

OD	Value Updates ($\times 10^6$)		Time Taken (s)	
	FP	UDP	FP	UDP
0	0.2	0.4	0.2	0.3
5	0.2	0.3	0.2	0.3
10	0.2	0.4	0.2	0.4
15	0.2	0.8	0.2	0.9
20	1.0	2.5	1.3	3.0

Table 2. Computation results from unfocussed FP using a Δ threshold of 10^{-3}. The focussed FP results from the first experiment are included for comparison.

der of their $\mathcal{G}(s, g)$ values. Since the convergence of value iteration does not depend on the order in which states are processed, we can use the ordering imposed by this unfocussed version of FP without jeopardising convergence. Interestingly, even when we remove the focussing aspect of FP we are still able to generate results more efficiently than the other approaches compared here. Table 2 shows the results obtained using unfocussed FP.

7. Conclusion

In this paper we have presented *Focussed Dynamic Programming*, an algorithm that efficiently solves restricted Markov decision processes. The approach uses heuristics to both focus computation towards relevant areas of the state space and to update state values in a sensible order. We have presented extensive comparisons between our algorithm and several current approaches applied to a robotic path planning domain. These results show that our approach is significantly more efficient than existing methods for generating suboptimal solutions to such MDPs.

We have also provided an unfocussed version of the algorithm which converges to the optimal value function over the entire state space. The results for this approach highlight that, even without focussing our computation towards the start state, the use of heuristic value estimates provides remarkable savings in processing time.

We believe that the principles behind this algorithm are applicable in many domains besides restricted Markov decision processes. The notion of prioritized state expansion through the use of heuristics has been a core component of deterministic planning, and, as shown in this paper, it can be applied successfully to stochastic planning.

Acknowledgements

This work was sponsored by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance" (contract number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the U.S. Government.

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, 72:81–138, 1995.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] D. Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, 1976.

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

[5] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[6] B. Bonet and H. Geffner. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, Italy, 2003.

[7] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence*, 76(1–2):35–74, July 1995.

[8] Z. Feng and E. Hansen. Symbolic Heuristic Search for Factored Markov Decision Processes. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Edmonton, Canada, 2002.

[9] D. Ferguson and A. Stentz. Focussed Dynamic Programming: Extensive Comparative Results. Technical Report CMU-RI-TR-04-13, Carnegie Mellon Robotics Institute, March 2004.

[10] E. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[11] R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Massachusetts, 1960.

[12] P. Laroche. Building Efficient Partial Plans using Markov Decision Processes. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2000.

[13] A. Moore and C. Atkeson. Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning*, 13:103–130, 1993.

[14] N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[15] L. Ojeda and J. Borenstein. FLEXnav: Fuzzy logic expert rule-based position estimation for mobile robots on rugged terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[16] M. Trick and S. Zin. Spline approximations to value functions: a linear programming approach. *Macroeconomic Dynamics*, 1:255–277, 1997.