Focussed Dynamic Programming: Extensive Comparative Results

Dave Ferguson Anthony Stentz

CMU-RI-TR-04-13

March 2004

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

© Carnegie Mellon University

Abstract

We present a heuristic-based propagation 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 in producing a solution. We present a number of results comparing our approach to existing algorithms on a robotic path planning domain.

Contents

1 Introduction **1**

2 Markov Decision Processes **1**
2.1 Path Planning MDPs 1
2.2 Classical Approaches 2

3 Related Approaches **3**
3.1 Real-Time Dynamic Programming 3
3.2 Envelope Propagation 3
3.3 LAO* 4
3.4 Prioritized Sweeping 4

4 Focussed Dynamic Programming **5**
4.1 Actuation Uncertainty in Path Planning MDPs 6

5 Results **9**

6 Discussion **16**

7 Conclusion **19**

1 Introduction

Markov decision processes (MDPs) have been widely used as a model 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 burdensome. As a result, a number of researchers have investigated ways of reducing this computation, by restricting the number of states considered or processing the states in a particular order.

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

We begin by reviewing MDPs and classical techniques for solving them. In Section 3, a number of more recent approaches are described, which 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. Finally, we conclude in Section 6 with additional discussion.

2 Markov Decision Processes

A Markov decision process is defined as a tuple (S, A, P, R), where S is the set of states of the world, 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 $R : S \times A$ is the reward function, where $R(s_i, a)$ specifies the expected reward for 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 reward function R, we can define the value of a state s_i to be its expected total (undiscounted) reward under the policy π. Given this definition, the value of state s_i given policy π can be written as [10]:

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

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

2.1 Path Planning MDPs

We are interested in solving the domain of robotic path planning. It is most often the case in path planning that we are trying to minimize the overall cost incurred along a path to the goal, rather than maximizing some reward function. Thus, instead of having

rewards associated with each state-action pair, we have costs $C : S \times S$ associated with moving between pairs of neighboring states. Since, in path planning, the states of the system typically correspond to positions in the environment, these costs are generally based on the difficulty of the terrain associated with the two states. The modified value equation becomes

$$V_\pi(s_i) = \sum_{s_j \in S} P(s_i, \pi(s_i), s_j) \cdot (C(s_i, s_j) + V_\pi(s_j)). \quad (1)$$

There are a few key properties of the 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. 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 motion models of current robotic actuators (see [14, 13]), 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 [10], or linear programming [3, 15]. 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, beginning from the start state and following 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

In [7], Dean et al. 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. Ten depth-first paths from the start state to the goal are generated. 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 tried 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, following the current policy. If a run encounters a state which 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 which 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 [9, 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, which represents all states adjacent to the current envelope which 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 [9]. 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 it 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 [11]. Rather than performing a number of passes through the environment in which every state has its value 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, then perform PS propagation from the start outwards. The second is to initialise all states (except for the goal) with infinite values and perform propagation 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 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 which 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 which 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 $s \in \{x \cup nbrs(x)\}$

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: The Focussed Dynamic Programming Algorithm

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 which 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 which have had their values updated towards areas which 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 the deterministic search algorithm Focussed Dynamic A* (D*) [12], 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

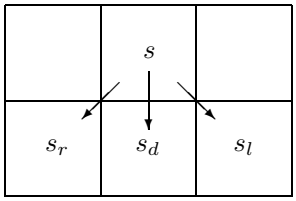


Figure 2: The possible resulting states after attempting to move from state s to state s_d using our action model.

of a state's value helps ensure that states that could have low optimal values but have high current values are updated, and incorporating the heuristic cost to the start state favors states which 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.

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 complete algorithm is given in Figure 1. Thus far, we have not mentioned how the heuristic functions $\mathcal{H}(r, s)$ and $\mathcal{G}(s, g)$ compute their values or what termination criteria we use. We discuss these choices with respect to our desired application domain.

4.1 Actuation Uncertainty in Path Planning MDPs

To take into account the uncertainty associated with current robotic actuators, we employ a motion model which assumes error in the translational and rotational directions (as in [14, 13]). Our planning is performed over an eight connected grid, with eight available actions from each state in the grid. Our motion model takes the shape of a mapping from states and actions to probability distributions over adjacent states.

Since the distance associated with each action is small, it is unrealistic to assume that an action may take an agent wildly off its intended track. As such, our motion model assigns nonzero probabilities to only three adjacent states for each action. Figure 2 shows the possible resulting states after attempting to move from state s to s_d. Of course, the probabilities associated with ending up in each of these states are different: it is much more likely that the agent will end up in state s_d than either of its unintended neighbors. Notice that we have not allowed for the possibility that the action takes the



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

agent backwards or directly sidewards. We are trying to model realistic robots and their actuation error over small periods is not nearly this extreme. However, in our discussion we do mention generalising the action 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.

Given this domain, we compute the heuristics $\mathcal{H}(r, s)$ and $\mathcal{G}(s, g)$ as follows. For our admissible heuristic function $\mathcal{H}(r, s)$, we use the standard Euclidean distance metric used in deterministic planning.

The calculation of the heuristic value of a state, $\mathcal{G}(s, g)$, is a little more complicated. In deterministic search algorithms such as D* 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 nondeterministic 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 three possible resulting states after performing action a (s_d, s_l, and s_r) 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 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 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* and D* 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 [9].

The termination criteria we use is that of D*: 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 results very close to optimal.

Figure 3 shows the states examined leading up to various stages of the algorithm when applied to a small example map. The start state is the blue (shaded) cell at the far left center; the goal is the blue (shaded) cell at the far right. Note that the red (dark gray) cells are the only cells which would be examined regardless of the size of the map.

Table 1: The number of updates and time taken by each algorithm to reach the same start error as Focussed Dynamic Programming.

OD	Average Number of Value Updates (times 10^6)								Average Time Taken (in seconds)								Error
	FP	VIO	VIS	EP	LAO	RT	LR	PS	FP	VIO	VIS	EP	LAO	RT	LR	PS	(%)
0	0.2	3.0	0.8	1.7	2.2	4.4	2.4	1.7	0.2	1.8	0.5	2.3	1.3	3.5	1.7	6.6	0.01
1	0.2	3.1	0.8	2.0	2.3	3.7	2.2	1.7	0.2	1.8	0.5	3.1	1.4	3.0	1.6	6.8	0.01
2	0.2	3.0	0.9	2.4	2.3	5.7	2.0	1.6	0.2	1.8	0.5	3.9	1.4	4.5	1.5	6.5	0.01
3	0.2	9.0	0.9	2.8	2.4	5.0	2.0	1.6	0.2	5.8	0.6	5.2	1.7	4.3	1.5	7.3	0.01
4	0.2	13.2	1.0	3.3	2.5	9.9	1.8	1.6	0.2	9.1	0.7	7.0	1.8	9.1	1.4	7.5	0.00
5	0.2	16.5	1.0	3.9	2.6	9.8	1.8	1.5	0.2	10.6	0.6	8.0	1.7	8.0	1.4	6.7	0.00
6	0.2	28.2	1.1	4.3	2.7	10.2	1.7	1.5	0.2	17.9	0.7	10.0	1.8	8.4	1.4	6.7	0.00
7	0.2	36.6	1.1	5.1	2.9	10.3	1.7	1.5	0.2	25.2	0.8	13.8	2.1	9.7	1.4	7.0	0.00
8	0.2	36.6	1.2	6.0	2.9	12.8	1.6	1.5	0.2	22.6	0.7	15.9	1.8	10.4	1.4	6.3	0.00
9	0.2	48.1	1.2	5.5	2.9	6.8	1.6	1.4	0.2	33.5	0.8	18.1	2.1	5.8	1.3	6.5	0.01
10	0.2	52.2	1.5	6.5	3.0	14.1	1.5	1.5	0.2	34.5	0.9	20.5	2.0	12.1	1.3	6.4	0.52
11	0.2	54.5	1.5	6.4	2.9	8.3	1.5	1.4	0.2	38.3	1.0	24.0	2.1	7.4	1.3	6.5	0.31
12	0.2	54.5	1.4	7.2	3.3	12.2	1.6	1.6	0.2	38.9	1.0	28.9	2.4	11.1	1.4	7.2	0.01
13	0.2	54.7	1.7	7.4	3.4	14.5	1.7	1.6	0.2	39.1	1.2	31.7	2.5	12.4	1.4	7.2	0.39
14	0.2	55.0	2.3	8.4	3.8	11.5	2.0	1.7	0.2	40.4	1.7	37.9	2.7	9.7	1.6	7.7	0.19
15	0.2	55.0	1.7	9.2	3.9	37.1	2.4	2.1	0.2	40.0	1.2	41.4	2.8	32.0	1.9	9.4	0.00
16	0.2	55.4	3.2	11.7	4.7	10.5	3.5	3.3	0.3	43.0	2.5	60.3	4.0	9.9	2.8	16.7	0.03
17	0.2	55.7	4.8	13.4	5.3	14.1	3.9	4.4	0.3	38.4	3.3	58.9	3.7	11.8	3.0	18.6	0.01
18	0.4	56.4	6.2	16.5	7.2	12.7	6.4	5.7	0.6	43.3	4.6	88.0	5.5	12.1	5.0	27.1	0.14
19	0.8	58.6	13.1	31.0	10.9	16.2	11.1	13.3	1.1	44.7	10.1	141	7.8	13.6	8.8	58.0	0.33
20	1.0	60.6	16.6	39.8	12.6	6.1	14.8	21.6	1.3	44.2	11.9	167	9.5	5.7	11.4	101	1.74

5 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 random environments, each of dimension 200×200. We varied the number of obstacle cells in the environments so that we had 20 maps for each obstacle density from 0 (no obstacle cells) to 20 (one cell out of every five is an obstacle). In each map, the terrain of non-obstacle cells was also randomly generated, and the cost of traversing between two adjacent cells was a function of their respective terrains. The start and goal states were the same for each environment: the start was at the center of the left edge of the environment with the goal at the center of the right edge. We assumed a symmetric error model for our actions, with $P(s, a, s_d) = 0.85$ and $P(s, a, s_l) = P(s, a, s_r) = 0.075$.

In the first experiment, we initially ran value iteration until it converged on an optimal value for each state in the world (reported as VIO in our results)[1]. We then

[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.

Table 2: The standard deviation associated with the number of updates and time taken by each algorithm to reach the same start error as Focussed Dynamic Programming.

OD	Standard Deviation of Value Updates (times 10^6)								Standard Deviation of Time Taken (in seconds)							
	FP	VIO	VIS	EP	LAO	RT	LR	PS	FP	VIO	VIS	EP	LAO	RT	LR	PS
0	0.0	0.1	0.1	0.1	0.1	2.1	0.4	0.2	0.0	0.0	0.0	0.2	0.0	1.6	0.2	0.5
1	0.0	0.1	0.0	0.2	0.1	1.9	0.3	0.1	0.0	0.0	0.0	0.3	0.1	1.4	0.2	0.5
2	0.0	0.1	0.1	0.3	0.1	5.5	0.4	0.1	0.0	0.0	0.0	0.4	0.1	4.2	0.2	0.5
3	0.0	12.8	0.1	0.7	0.1	3.7	0.3	0.1	0.0	8.7	0.1	0.9	0.1	3.1	0.2	0.7
4	0.0	15.4	0.1	0.6	0.2	10.1	0.2	0.1	0.0	12.1	0.2	1.3	0.4	9.4	0.1	1.5
5	0.0	17.2	0.1	0.8	0.2	7.9	0.2	0.1	0.0	11.3	0.1	1.6	0.1	6.3	0.1	0.5
6	0.0	22.3	0.1	0.8	0.2	10.5	0.2	0.1	0.0	13.7	0.1	2.0	0.2	8.2	0.1	0.9
7	0.0	20.0	0.1	0.9	0.2	10.2	0.1	0.1	0.0	13.5	0.1	2.6	0.3	10.3	0.1	0.7
8	0.0	19.7	0.1	1.6	0.2	14.9	0.1	0.1	0.0	12.1	0.1	3.3	0.2	11.8	0.1	0.5
9	0.0	14.5	0.1	1.0	0.2	7.9	0.1	0.1	0.0	11.2	0.1	4.0	0.3	6.1	0.1	0.5
10	0.0	9.1	0.8	2.1	0.5	13.8	0.2	0.2	0.1	6.5	0.5	4.3	0.4	12.1	0.2	0.9
11	0.1	1.7	0.8	1.3	0.3	8.5	0.2	0.2	0.1	3.2	0.6	4.3	0.3	7.3	0.1	0.9
12	0.0	2.0	0.1	1.6	0.4	12.8	0.3	0.1	0.0	3.7	0.1	5.2	0.3	11.9	0.2	0.7
13	0.1	2.8	0.8	2.3	0.7	11.4	0.4	0.2	0.1	3.4	0.7	7.8	0.5	9.7	0.3	1.0
14	0.1	2.8	2.8	1.7	1.2	13.9	0.5	0.3	0.1	8.1	1.8	9.1	0.8	11.4	0.3	1.2
15	0.0	2.5	0.2	2.1	0.6	51.9	1.0	0.3	0.0	4.5	0.1	8.1	0.5	44.0	0.8	1.8
16	0.1	2.6	4.1	6.4	1.6	9.2	1.4	3.1	0.2	7.9	2.9	23.2	1.8	8.1	1.1	15.3
17	0.1	2.3	5.3	7.6	1.8	18.3	1.0	3.5	0.2	1.8	3.6	22.8	1.2	15.2	0.8	15.1
18	0.3	3.2	4.6	7.2	2.6	14.3	2.1	3.1	0.5	10.4	3.3	32.6	2.2	14.0	1.6	16.8
19	0.5	3.9	6.1	13.6	3.3	50.1	3.1	6.7	0.7	12.6	5.3	71.7	2.3	41.1	3.1	28.3
20	0.7	4.0	4.9	22.1	4.4	4.1	5.0	9.0	0.9	5.1	3.3	53.9	3.4	4.1	3.9	44.4

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.

We have strived to represent each algorithm favorably, in some cases making fairly significant alterations in order to improve overall performance.

For Envelope Propagation, we performed several optimisations. Firstly, we performed a heuristic-based search to generate the initial envelope, rather than depth-first search. Secondly, during envelope expansion we added *all* states encountered that were not in the envelope, and added them immediately rather than waiting until all simulated runs were completed. This was because, especially towards the start of processing, the

runs would end up in the same state a large percent of the time, and this would limit the number of new elements added to the envelope. Thirdly, we performed value iteration rather than policy iteration to update values within the envelope.

Finally, we found, as did Hansen and Zilberstein [9], that both EP and LAO* performed much more favorably when the dynamic programming policy generation phase was not run to convergence. For our experiments, a single forwards and backwards sweep was most effective.

Together, these changes drastically improved the overall performance of the EP approach. We also experimented with assigning inadmissible values to states outside the envelope, but this resulted in very restricted envelopes that gave highly sub-optimal results (and terminated without approaching optimality).

With PS, we found that propagating inwards from the goal was much more efficient than propagating outwards from the start. This is primarily because the values of states near the goal can be quickly updated to be close to optimal, which allows for more accurate value updates of their neighbors. Further, this approach allowed us to restrict the set of states added to the queue to be only those whose admissible heuristic costs to the goal and to the start combined were less than the current value of the start state. When propagating out from the start, all values were admissible (including the value of the start state), so we could not use this limiting criteria. We have included in our initial results only the results for the second PS approach. Comparative results between the first and the second approach can be found in Table 5.

The results for the first experiment are in Tables 1 and 2 . The leftmost column represents the obstacle density (OD) of each set of environments. In the first table, 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. A graph illustrating the time taken by the four most efficient approaches is provided in Figure 4.

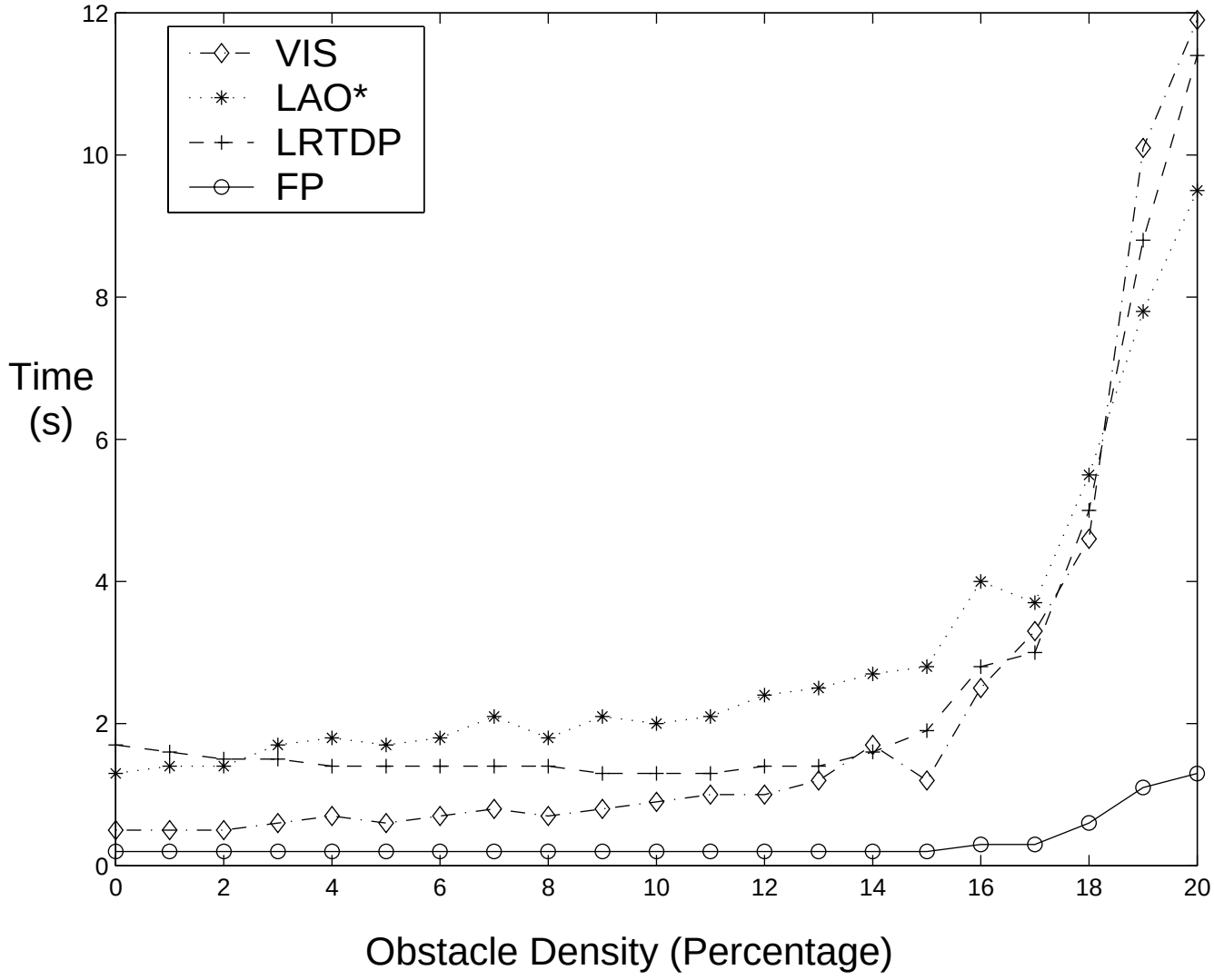


Figure 4: Run time required for the four most efficient approaches to get within the same error (for the start state value) as Focussed Dynamic Programming.

Table 3: The number of updates and time taken by each algorithm to get the value of the start within $\delta = 0.1$ of its optimal value.

OD	Average Number of Value Updates (times 10^6)							Average Time Taken (in seconds)						
	FP	VIS	EP	LAO	RT	LR	PS	FP	VIS	EP	LAO	RT	LR	PS
0	0.2	0.7	1.6	2.1	2.2	1.7	1.3	0.2	0.4	2.3	1.3	1.8	1.3	5.6
1	0.2	0.7	2.0	2.2	2.2	1.7	1.3	0.2	0.4	3.1	1.4	1.8	1.3	5.7
2	0.2	0.7	2.1	2.3	2.2	1.6	1.3	0.2	0.4	3.6	1.4	1.8	1.3	5.6
3	0.2	0.8	2.6	2.4	2.3	1.6	1.3	0.2	0.5	5.1	1.6	2.1	1.3	6.2
4	0.2	0.8	2.9	2.5	2.2	1.6	1.3	0.2	0.6	6.6	1.8	2.2	1.3	6.5
5	0.2	0.9	3.6	2.5	2.2	1.6	1.2	0.2	0.5	7.9	1.7	1.9	1.3	5.5
6	0.2	0.9	3.8	2.6	2.2	1.5	1.2	0.2	0.6	9.1	1.7	1.9	1.3	5.5
7	0.2	1.0	4.5	2.8	2.2	1.5	1.2	0.2	0.7	13.0	2.0	2.1	1.3	5.8
8	0.2	1.0	4.8	2.8	2.1	1.5	1.2	0.2	0.6	13.7	1.8	1.9	1.3	5.2
9	0.2	1.0	5.3	2.9	2.1	1.5	1.2	0.2	0.7	18.1	2.0	2.0	1.3	5.6
10	0.2	1.4	5.9	3.1	2.1	1.5	1.2	0.2	0.9	19.8	2.1	1.9	1.3	5.4
11	0.2	1.4	6.2	3.0	2.0	1.5	1.2	0.2	1.0	24.6	2.2	1.9	1.3	5.7
12	0.2	1.2	6.7	3.2	2.0	1.6	1.2	0.2	0.9	28.4	2.4	2.0	1.4	6.0
13	0.2	1.6	7.0	3.5	2.0	1.7	1.3	0.2	1.2	31.4	2.6	2.0	1.4	6.7
14	0.2	2.3	7.8	3.9	2.1	1.9	1.5	0.2	1.6	35.0	2.7	1.9	1.6	6.7
15	0.2	1.5	8.0	3.8	2.0	2.4	1.6	0.2	1.1	39.1	2.8	1.9	2.0	7.7
16	0.2	3.1	11.5	4.7	2.3	3.5	2.8	0.3	2.5	58.3	3.8	2.2	2.7	14.7
17	0.2	4.3	12.6	5.1	2.4	3.9	3.6	0.3	2.9	57.0	3.5	2.1	3.0	15.9
18	0.4	6.0	16.0	7.2	2.7	6.5	4.8	0.6	4.4	81.7	5.5	2.6	5.1	23.4
19	0.9	13.1	29.9	10.9	3.9	11.9	13.0	1.1	9.6	134.4	8.4	3.8	9.3	59.9
20	1.1	17.9	43.7	13.5	5.2	14.0	23.8	1.5	13.6	171.1	10.0	4.8	10.8	108.7

The results for the second experiment are in Tables 3 and 4. For this experiment, we set $\delta = 0.1$ (where optimal values for the start state in each environment ranged from 250 to 900). We have also included a graph showing the time performance of the five most efficient approaches for this experiment (see Figure 5).

Table 4: The standard deviation associated with the number of updates and time taken by each algorithm to get the value of the start within $\delta = 0.1$ of its optimal value.

OD	Standard Deviation of Updates (times 10^6)							Standard Deviation of Time Taken (in seconds)						
	FP	VIS	EP	LAO	RT	LR	PS	FP	VIS	EP	LAO	RT	LR	PS
0	0.0	0.0	0.1	0.1	0.0	0.1	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.3
1	0.0	0.0	0.2	0.1	0.1	0.1	0.1	0.0	0.0	0.3	0.0	0.1	0.1	0.3
2	0.0	0.0	0.3	0.1	0.1	0.1	0.1	0.0	0.0	0.4	0.1	0.1	0.1	0.2
3	0.0	0.0	0.4	0.1	0.1	0.1	0.1	0.0	0.0	0.7	0.1	0.2	0.1	0.4
4	0.0	0.0	0.4	0.2	0.1	0.1	0.0	0.1	0.1	1.7	0.4	0.5	0.0	1.3
5	0.0	0.0	0.7	0.2	0.1	0.1	0.0	0.0	0.0	1.4	0.2	0.1	0.1	0.4
6	0.0	0.0	0.6	0.2	0.1	0.1	0.1	0.0	0.1	2.1	0.2	0.2	0.1	0.7
7	0.0	0.0	0.6	0.2	0.1	0.0	0.1	0.0	0.1	2.0	0.2	0.2	0.0	0.6
8	0.0	0.1	0.8	0.2	0.2	0.1	0.1	0.0	0.1	2.1	0.2	0.2	0.1	0.3
9	0.0	0.0	1.0	0.2	0.1	0.1	0.0	0.0	0.1	4.2	0.3	0.2	0.1	0.5
10	0.1	1.5	1.0	0.6	0.2	0.1	0.1	0.1	0.9	3.2	0.4	0.2	0.1	0.5
11	0.1	1.4	0.9	0.5	0.2	0.2	0.1	0.1	1.1	3.7	0.4	0.2	0.1	0.5
12	0.0	0.1	1.3	0.4	0.2	0.3	0.1	0.0	0.1	4.9	0.3	0.2	0.2	0.7
13	0.1	1.5	1.2	0.6	0.2	0.3	0.1	0.1	1.2	6.5	0.6	0.4	0.2	1.4
14	0.1	3.0	1.6	1.2	0.3	0.4	0.1	0.1	2.0	5.8	0.8	0.2	0.3	0.6
15	0.0	0.1	1.9	0.6	0.2	1.1	0.2	0.0	0.1	6.2	0.5	0.3	0.8	1.4
16	0.1	4.3	6.9	2.0	0.6	1.6	3.1	0.3	3.8	27.3	2.2	0.8	1.2	16.7
17	0.1	4.7	6.9	1.5	0.5	1.3	3.1	0.2	3.2	21.2	1.0	0.5	1.0	13.9
18	0.3	4.9	7.9	2.5	0.6	2.7	2.6	0.5	3.6	29.6	1.8	0.6	2.1	13.4
19	0.5	6.1	13.6	3.3	0.9	3.8	6.9	0.7	4.6	87.8	4.1	1.7	3.0	29.9
20	0.7	4.7	22.7	4.0	1.8	4.2	9.1	0.8	3.9	57.8	3.4	1.8	3.3	43.3

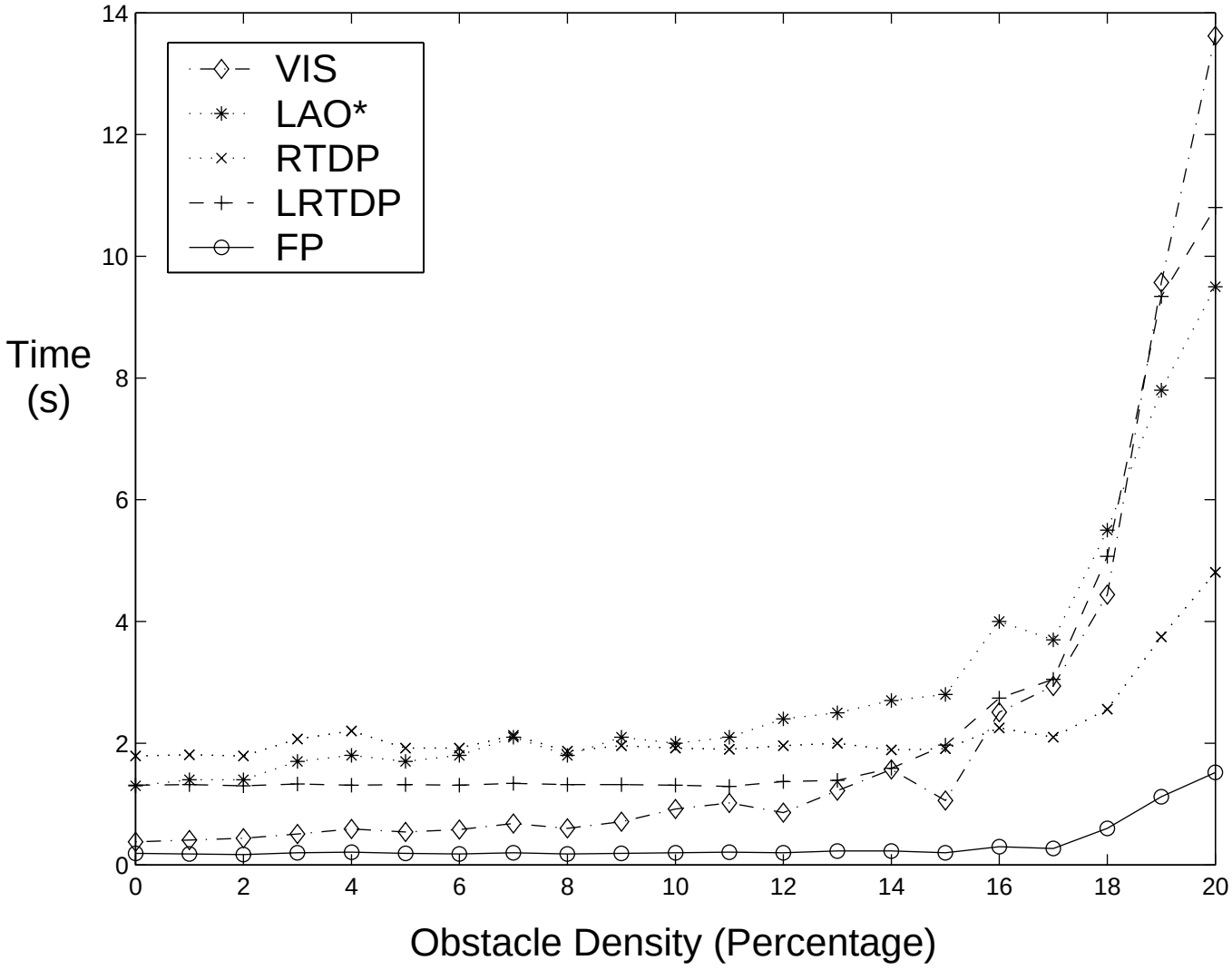


Figure 5: Run time required for the five most efficient approaches to get within 0.1 of the optimal value for the start state.

Table 5: Comparison between standard PS approach (PS1) and modified PS (PS2).

OD	Experiment 1				Experiment 2			
	Updates (10^6)		Time (s)		Updates (10^6)		Time (s)	
	PS1	PS2	PS1	PS2	PS1	PS2	PS1	PS2
0	5.3	1.7	20.8	6.6	5.3	1.3	20.7	5.6
1	5.1	1.7	19.8	6.8	5.1	1.3	19.8	5.7
2	5.1	1.6	19.7	6.5	5.1	1.3	19.8	5.6
3	4.9	1.6	21.1	7.3	4.9	1.3	21.2	6.2
4	4.9	1.6	19.0	7.5	4.9	1.3	19.0	6.5
5	4.8	1.5	18.4	6.7	4.8	1.2	18.4	5.5
6	4.7	1.5	18.1	6.7	4.7	1.2	18.1	5.5
7	4.7	1.5	19.5	7.0	4.7	1.2	19.6	5.8
8	4.7	1.5	19.6	6.3	4.7	1.2	19.3	5.2
9	4.8	1.4	18.1	6.5	4.8	1.2	18.1	5.6
10	4.9	1.5	18.5	6.4	4.9	1.2	18.5	5.4
11	5.1	1.4	19.7	6.5	5.1	1.2	19.7	5.7
12	5.4	1.6	20.7	7.2	5.4	1.2	20.6	6.0
13	6.2	1.6	23.3	7.2	6.2	1.3	23.3	6.7
14	7.5	1.7	28.1	7.7	7.5	1.5	28.1	6.7
15	8.7	2.1	32.1	9.4	8.7	1.6	32.2	7.7
16	12.9	3.3	47.3	16.7	12.9	2.8	47.4	14.7
17	15.3	4.4	56.7	18.6	15.2	3.6	56.7	15.9
18	22.4	5.7	83.8	27.1	21.5	4.8	80.0	23.4
19	33.7	13.3	121.6	58.0	33.2	13.0	120.6	59.9
20	48.8	21.6	191.7	101	46.0	23.8	177.2	108.7

Table 6: Example results using an action model with 5 possible outcomes for each action.

OD	Number of Value Updates (times 10^6)					
	FP	VIS	EP	LAO	RT	PS
9	2.5	19.9	136.5	167.1	5.8	14.6

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 nature of the envelope. For the current experiments, the start and goal states were on opposite ends of

Table 7: Computation results from unfocussed FP using a Δ threshold of 10^{-3}. The focussed FP results from the first experiment are included for comparison.

OD	Value Updates (times 10^6)		Time Taken (in seconds)	
	Focussed FP	Unfocussed FP	Focussed FP	Unfocussed FP
0	0.2	0.4	0.2	0.3
1	0.2	0.4	0.2	0.3
2	0.2	0.3	0.2	0.3
3	0.2	0.3	0.2	0.3
4	0.2	0.3	0.2	0.3
5	0.2	0.3	0.2	0.3
6	0.2	0.3	0.2	0.4
7	0.2	0.3	0.2	0.4
8	0.2	0.4	0.2	0.4
9	0.2	0.4	0.2	0.4
10	0.2	0.4	0.2	0.4
11	0.2	0.4	0.2	0.5
12	0.2	0.5	0.2	0.5
13	0.2	0.5	0.2	0.6
14	0.2	0.6	0.2	0.7
15	0.2	0.8	0.2	0.9
16	0.2	1.0	0.3	1.1
17	0.2	1.2	0.3	1.4
18	0.4	1.6	0.6	1.9
19	0.8	2.0	1.1	2.4
20	1.0	2.5	1.3	3.0

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 Table 6 for a sample result). However, we are of the opinion that this added complexity is unjustified for robotic path planning.

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 [9]. 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 order 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 other approaches. Table 7 shows the rsults 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 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 have shown our approach to provide significant benefit over existing methods.

We are currently investigating how we can use the ideas described here to perform efficient replanning with MDPs. This would be very useful for robots navigating through partially known environments.

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 Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, Edmonton, Canada, 2002.

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

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

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

[12] Anthony Stentz. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[13] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva. *International Journal of Robotics Research*, 19(11):972–999, 2000.

[14] S. Thrun, W. Burgard, and D. Fox. A Real-Time Algorithm for Mobile Robot Mapping With Applications to Multi-Robot and 3D Mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, 2000. IEEE.

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