Solving Uncertain MDPs by Reusing State Information and Plans

Ping Hou
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
phou@cs.nmsu.edu

William Yeoh
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
wyeoh@cs.nmsu.edu

Tran Cao Son
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
tson@cs.nmsu.edu

Abstract

While MDPs are powerful tools for modeling sequential decision making problems under uncertainty, they are sensitive to the accuracy of their parameters. MDPs with uncertainty in their parameters are called Uncertain MDPs. In this paper, we introduce a general framework that allows off-the-shelf MDP algorithms to solve Uncertain MDPs by planning based on currently available information and replan if and when the problem changes. We demonstrate the generality of this approach by showing that it can use the VI, TVI, ILAO*, LRTDP, and UCT algorithms to solve Uncertain MDPs. We experimentally show that our approach is typically faster than replanning from scratch and we also provide a way to estimate the amount of speedup based on the amount of information being reused.

Introduction

Markov Decision Processes (MDPs) are very rich models that can fully capture the stochasticity present in many sequential decision making problems. However, the parameters of these models, such as cost and transition functions, are often derived from domain experts or estimated from data. In both cases, these parameters can change as more experts provide input or more data is made available. The change in these parameters can significantly degrade the quality of policies found for the previous set of parameters (Mannor et al. 2007). These problems with uncertainty in the parameters as called *Uncertain MDPs*.

Generally, researchers have taken *proactive approaches* to solve Uncertain MDPs by explicitly representing the cost and transition functions as fixed but unknown parameters. Some assume that the parameters are elements of a known bounded set, called the uncertainty set, and use robust optimization techniques to solve the Uncertain MDPs (Givan, Leach, and Dean 2000; Iyengar 2005; Nilim and Ghaoui 2005; Regan and Boutilier 2009; 2011; Mannor, Mebel, and Xu 2012). Alternatively, some assume that the parameters are random variables that follow some distribution and use percentile optimization techniques (Delage and Mannor 2010), distributional robustness (Xu and Mannor 2012), or sampling-based approaches (Ahmed et al. 2013) to solve the Uncertain MDPs.

Another class of approaches are *reactive approaches* like incremental search (Stentz 1995; Koenig and Likhachev 2002; Koenig et al. 2004; Likhachev, Gordon, and Thrun 2003; Sun, Koenig, and Yeoh 2008; Sun, Yeoh, and Koenig 2009; 2010a; 2010b), which is thus far restricted to problems with deterministic transitions. Incremental search algorithms typically use A* (Hart, Nilsson, and Raphael 1968) to find a plan for the initial problem, and replans each time the problem changes. It uses different techniques to identify parts of the previous plan that are unaffected by the change in the problem and reuses them to replan for the new problem. Researchers have shown that by reusing parts of the previous plan, the replanning process can be sped up significantly especially if the changes are small.

One common theme across most of the existing proactive approaches is the assumption that it is possible to model the uncertainty in the parameters of the MDP. However, it can be difficult to ascertain if the model is accurate with a high degree of confidence. For example, the parameters of the distribution might not be known. On the other hand, most of the reactive approaches are thus far limited to deterministic problems – mostly agent-based path-planning problems.

Our goal in this paper is to find a balance between both approaches. We introduce a general *Incremental MDP Framework* that allows standard off-the-shelf MDP algorithms to solve Uncertain MDPs by planning based on currently available information (modeled as an initial MDP) and replan if and when the problem changes (modeled as a new MDP). Like incremental search algorithms, this framework endeavors to speed up the planning process by identifying parts of the previous plan that are reusable in the current planning process. Therefore, unlike the proactive approaches, we do not require a model of the uncertainty in the parameters of the MDP model, and unlike the reactive approaches, we are not limited to deterministic problems. We demonstrate the generality of this approach by showing that it can use several off-the-shelf algorithms to solve Uncertain MDPs. We experimentally show that our approach is typically faster than replanning from scratch and finds better solutions when given a fixed runtime.

MDP Model

A *Stochastic Shortest Path Markov Decision Process* (SSP-MDP) (Bertsekas 2000) is represented as a tuple

$\langle \mathbf{S}, s_0, \mathbf{A}, \mathbf{T}, \mathbf{C}, \mathbf{G} \rangle$. It consists of a set of states $\mathbf{S}$; a start state $s_0 \in \mathbf{S}$; a set of actions $\mathbf{A}$; a transition function $\mathbf{T} : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \rightarrow [0, 1]$ that gives the probability $T(s, a, s')$ of transitioning from state s to s' when action a is executed; a cost function $\mathbf{C} : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \rightarrow \mathbb{R}$ that gives the cost $C(s, a, s')$ of executing action a in state s and arriving in state s'; and a set of goal states $\mathbf{G} \subseteq \mathbf{S}$. The goal states are terminal, that is, $T(g, a, g) = 1$ and $C(g, a, g) = 0$ for all goal states $g \in \mathbf{G}$ and actions $a \in \mathbf{A}$. An SSP-MDP must also satisfy the following two conditions: (1) There must exist a *proper policy*, which is a mapping from states to actions with which an agent can reach a goal state from any state with probability 1. (2) Every *improper policy* must incur an accumulated cost of ∞ from all states from which it cannot reach the goal with probability 1. In this paper, we will focus on SSP-MDPs and will thus use the term MDPs to refer to SSP-MDPs. However, our techniques also apply to regular finite-horizon and infinite-horizon discounted-reward MDPs as well since they are subclasses of SSP-MDPs (Bertsekas and Tsitsiklis 1996).

The state space of an MDP can be visualized as a directed hyper-graph called a *connectivity graph*. Figure 1 shows an example partial connectivity graph, where states of the MDP correspond to nodes (denoted by circles) and transitions between states correspond to hyper-edges (denoted by arrows). The subgraph rooted at a start state is called a *transition graph*. It is possible to partition the connectivity or transition graphs into *Strongly Connected Components* (SCCs) in such a way that they form a *Directed Acyclic Graph* (DAG) (Tarjan 1972; Bonet and Geffner 2003a; Dai et al. 2011). Thus, it is impossible to transition from a state in a downstream SCC to a state in an upstream SCC. Each SCC is denoted by a rectangle in Figure 1. We call this DAG an *SCC transition tree*.

MDP Algorithms

An MDP policy $\pi : \mathbf{S} \rightarrow \mathbf{A}$ is a mapping from states to actions. Solving an MDP is to find an optimal policy, that is, a policy with the smallest expected cost. Generally, there are two classes of MDP algorithms with different representations of expected costs and action selections. The first class of algorithms uses a value function V to represent expected costs. The expected cost of an optimal policy π^* is the expected cost $V(s_0)$ for starting state s_0, which is calculated using the Bellman equation (Bellman 1957):

$$V(s) = \min_{a \in \mathbf{A}} \sum_{s' \in \mathbf{S}} T(s, a, s') \big[C(s, a, s') + V(s') \big] \qquad (1)$$

The action chosen for the policy for each state s is then the one that minimizes $V(s)$. A single update using this equation is called a *Bellman update*. The second class of algorithms uses Q-values $Q(s, a)$ to represent the expected cost of taking action a in state s. The action chosen for the policy for each state s is then the one that minimizes $Q(s, a)$.

We now briefly describe five common MDP algorithms, which we can use as subroutines in our algorithm. The first four are algorithms of the first class and the last one is an algorithm of the second class. We refer the reader to the original papers for more details.

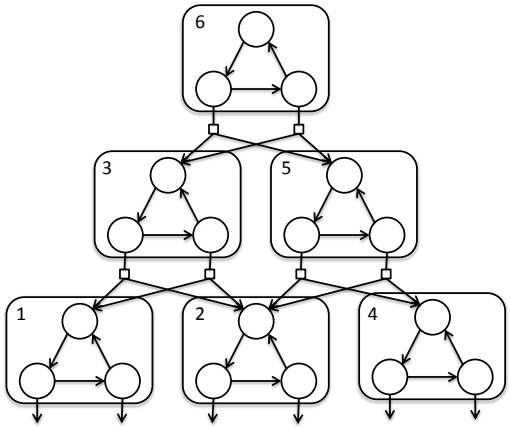

Figure 1: Partial Connectivity Graph

Value Iteration (VI): VI (Bellman 1957) is one of the fundamental approaches to find an optimal policy. In each iteration, it performs a Bellman update on each state. The difference between the expected cost of a state in two consecutive iterations is called the residual of that state and the largest residual is called the *residual error*. The algorithm terminates when the values converge, that is, the residual error is less than a user-defined threshold ϵ. Lastly, VI can be optimized by only considering the set of states reachable from the start state.

Topological VI (TVI): VI suffers from a limitation that it updates each state in every iteration even if the expected cost of states that it can transition to remain unchanged. TVI (Dai et al. 2011) addresses this limitation by repeatedly updating the states in only *one* SCC until their values converge before updating the states in another SCC. Since the SCCs form a directed acyclic graph, states in an SCC only affect the states in upstream SCCs. Thus, by choosing the SCCs in reverse topological sort order, it no longer need to consider SCCs whose states have converged in a previous iteration. Figure 1 shows the indices of the SCCs in reverse topological sort order on the upper left corner of the squares. Like VI, TVI can also be optimized by only considering the set of states reachable from the start state.

Improved LAO* (ILAO*): ILAO* (Hansen and Zilberstein 2001) is another approach that uses heuristics to improve its search process. It builds an explicit transition graph, where nodes correspond to states and directed edges correspond to transitions between states. It starts with the start state as the sole node in the graph. Then, in each iteration, (1) it expands all the fringe states in the greedy policy graph, which is a subgraph that is formed by taking a greedy policy on the values of the states in the original graph; and (2) it performs a single Bellman update for each state in the graph after expanding all the fringe states. The exception is when it expands a *reusable* state, in which case it expands all the states in the subtree rooted at that state before performing the Bellman updates. This exception is only necessary when ILAO* is used in conjunction with our framework, which labels states as reusable or non-reusable. We describe this

framework later. This expansion process continues until the graph has no more non-terminal fringe states, at which point ILAO* performs repeated Bellman updates until the values converge, that is, the residual error is less than a user-defined threshold ϵ.

Labeled Real-Time Dynamic Programming (LRTDP): One of the limitations of VI, TVI, and ILAO* is that they might take a long time to find a policy. In such situations, anytime algorithms like LRTDP (Bonet and Geffner 2003b) are desirable. As long as there is time, LRTDP simulates the current greedy policy to sample trajectories (called a *trial* in LRTDP terminology) through the state space and performs Bellman updates only on the states along those trajectories. These updates change the greedy policy and make way for further improvements on the states' expected costs. In each trial, starting from the start state, a greedy action is selected based on the current expected costs and the resulting state of this action is sampled stochastically. This process ends when a goal state or a *solved* state is reached. A state is considered solved if the residual is less than a user-defined threshold ϵ for every state in the transition graph rooted at the solved state and is reachable with the greedy policy.

UCB applied to Trees (UCT): Aside from LRTDP, another anytime algorithm is a sampling-based algorithm called UCT (Kocsis and Szepesvári 2006). Like LRTDP, it samples trajectories (called a *rollout* in UCT terminology) through the state space as well. However, it uses Q-values instead of value functions to represent expected costs. It repeatedly updates $Q(s,a)$ so that the Q-value is the average cost accumulated in past rollouts when it executes action a in state s. UCT selects an action a' using the following equation to balance exploration and exploitation:

$$a' = \arg\min_{a \in \mathbf{A}} \left\{ Q(s,a) - B\sqrt{\frac{\ln\left(\sum_{a''} n_{s,a''}\right)}{n_{s,a}}} \right\} \qquad (2)$$

where B is a constant and $n_{s,a}$ is the number of times the rollout choose action a in state s.

Incremental MDP Framework

We now describe the high-level ideas of our framework. This framework capitalizes on a key observation: States in downstream SCCs can not transition to states in upstream SCCs because the SCC transition tree is a directed acyclic graph. As a result, the policy and state information in upstream SCCs depend on the policy and state information in downstream SCCs, but not vice versa. Thus, if there are changes in the problem that affects the correctness of the policy and state information of some states (we call such states *affected states* or, equivalently, *non-reusable states*), then this error propagates only to other states in the same SCC and upstream SCCs. The error does not propagate to states in downstream SCCs and, thus, those states are *unaffected states* or, equivalently, *reusable states*. By reusing the state information in reusable states to find a new policy for the updated problem, we postulate that (1) when given a desired solution quality, our approach should be able to find such a solution faster than solving the problem from scratch; and (2) when given the same amount of runtime, our approach should be able to find better solutions than those found by solving the problem from scratch.

Algorithm 1: INCREMENTAL-MDP(M)

```
1  I = INITIALIZE(M)
2  (π, I) = MDP-SOLVER(M, I)
3  s = s_0
4  while s ∉ G do
5      s = EXECUTE-AN-ACTION(M, π, s)
6      Δ = DETECT-CHANGES()
7      if Δ ≠ NULL then
8          M = UPDATE-MODEL(M, Δ)
9          S_Δ = FIND-AFFECTED-SOURCES(M, Δ)
10         Y = FIND-SCCS(M, s)
11         I = FIND-REUSABLE-STATES(I, S_Δ, Y)
12         I = UPDATE-INFORMATION(M, I, s)
13         (π, I) = MDP-SOLVER(M, I)
14     end
15 end
```

Algorithm 2: FIND-REUSABLE-STATES(I, S_Δ, Y)

```
16 for SCCs y_i with indices i = 1 to n do
17     if S_Δ ∩ y_i = {} then
18         y_i.reusable = true
19     else
20         y_i.reusable = false
21     end
22 end
23 for SCCs y_i with indices i = 1 to n do
24     if ∃ SCC y ∈ Succ(y_i) : ¬y.reusable then
25         y_i.reusable = false
26     end
27     for states s ∈ y_i do
28         s.reusable = y_i.reusable
29     end
30 end
31 return I
```

Pseudocode

Algorithm 1 shows the pseudocode of this framework. To illustrate the generality of this framework, we describe the operation of this framework for all the 5 MDP solvers described earlier. Given an initial MDP model M, it first initializes the information of each state with initial heuristic values (Line 1). The information of each state s consists of the flag $s.reusable$ (for all algorithms); the value function $V(s)$ and residual $s.residual$ (for VI, TVI, ILAO*, and LRTDP); the flag $s.solved$ (for LRTDP); and the Q-values $Q(s,a)$ and sampling frequency $n_{s,a}$ (for UCT). The flag $s.reusable$ indicates whether state s is reusable or, equivalently, unaffected by the changes in the problem.

Next, the framework can use any of the 5 MDP solvers described earlier to solve the MDP and find policy π and

updated state information I (Line 2). The algorithm then repeats the following steps until the agent reaches a goal state (Line 4): It executes an action according to policy π, moves to a new state s (Line 5), and checks if there are any changes Δ that need to be made to the model (Line 6). A change in the model is necessary if domain experts provide updated information or if the agent detects inconsistencies between the model and the environment. For example, the agent might detect an obstacle between states s and s' that prohibits it from moving between the two states but there exists an action a with transition $T(s, a, s') > 0$ in the MDP model.

If there are changes that need to be made (Line 7), then the agent updates its MDP model based on these changes (Line 8) and finds the immediate states that are affected by these changes (Line 9). Then, it partitions the state space of the new MDP model into SCCs (Line 10); finds the reusable states (Line 11), which are states that are unaffected by the changes Δ; and reinitializes the information of non-reusable states (Line 12). Lastly, it solves the updated MDP model M with state information I to find the new policy π (Line 13). We now describe some of these functions.

Detect-Changes: The possible types of changes are addition/removal of states in $\mathbf{S}$, changes in the start state s_0, addition/removal of actions in $\mathbf{A}$, changes in the transition function $\mathbf{T}$, changes in the cost function $\mathbf{C}$, and the addition/removal of goal states in $\mathbf{G}$. However, an important requirement is that the resulting MDP after the change is still an SSP-MDP.

Changes in the start state are included in the set of changes Δ only if the change is a result of exogenous factors and not a result of action execution. In other words, the new start state s_0 is included only if it is not one of the successor states of the previous start state s_0' with the previous action execution a, that is, $T(s_0', a, s_0) = 0$. The exception is if VI or TVI is used and if the new start state is in the subgraph rooted at the previous start state since they must have previously found a policy for all states in this subgraph. It is also important to note that if the *only* change in Δ is the change in the start state, then there is no need to run Lines 9-12 because all state information is still reusable.

Some of the changes can be represented by changes in the transition and cost functions alone. They are the removal of states s (represented by setting $T(s', a, s) = 0$ for all states s' and actions a) and the removal of actions a from state s (represented by setting $T(s, a, s') = 0$ for all states s'). Lastly, the remaining changes must include changes to the transition and cost functions. They are the addition of states s (includes adding $T(s', a, s)$, $T(s, a, s')$, $C(s', a, s)$, and $C(s, a, s')$ for all states s' and actions a), the addition of actions a (includes adding $T(s, a, s')$ and $C(s, a, s')$ for all states s and s'), the addition of goal states g (includes setting $T(g, a, g) = 1$ and $C(g, a, g) = 0$ for all actions a) and removal of goal states g (includes setting $T(g, a, g) \neq 1$).

Find-Affected-Sources: Given the set of changes Δ, the function returns the set of affected source states S_Δ. A state s is an *affected source state* iff its transition function $T(s, a, s')$ or cost function $C(s, a, s')$ is newly added or has changed for any action a and state s'.

Find-SCCs: One can partition the state space into SCCs with Tarjan's algorithm (Tarjan 1972), which traverses the connectivity graph in a depth-first manner and marks the SCC membership of each state. Tarjan's algorithm returns an SCC transition tree Y, where the SCC indices are in reverse topological sort order. One can optimize the algorithm by executing this function only in the first iteration and only when the underlying transition graph changes in future iterations. The underlying transition graph changes when there exists at least one transition function that changes from zero to non-zero or vice versa.

Find-Reusable-States: Given the set of affected source states S_Δ and the SCC transition tree Y, the algorithm determines whether each state s in the transition graph is affected by the changes Δ and sets the flag of those states $s.reusable$ accordingly. Algorithm 2 shows the pseudocode. It first determines if each SCC contains an affected source state and is thus not reusable (Lines 16-22). Then, it determines if each SCC is affected and is thus not reusable. An SCC is affected if it contains an affected source state or one of its downstream SCCs contains an affected source state (Lines 23-26). Once every SCC is determined to be reusable or not, the function sets the flag $s.reusable$ of each state s to true if its SCC is reusable and false otherwise (Lines 27-29). Finally, the function returns the state information with the updated reusable flags (Line 31).

Update-Information: Given the state information I with updated reusable flags, the function re-initializes all the information of all non-reusable states. More precisely, it reinitializes the value function $V(s)$ and residuals $s.residual$ (for VI, TVI, ILAO*, and LRTDP), the flag $s.solved$ (for LRTDP) and the Q-value $Q(s, a)$ and sampling frequency $n_{s,a}$ (for UCT) for each non-reusable state s.[1]

Theoretical Properties

Definition 1 *A state is non-reusable iff its state information is no longer correct due to changes in the MDP.*

Lemma 1 *A state is non-reusable iff it belongs to either an SCC that contains an affected source state or an SCC that is upstream of an SCC that contains an affected source state.*

Proof Sketch: For algorithms that are based on Bellman updates like VI, TVI, ILAO*, and LRTDP, the value function $V(s)$ of a state s is no longer correct in the following cases:[2]

Case 1: It is an affected source state. The reason is that the newly added or changed transition or cost function can change $V(s)$ according to Equation 1.

Case 2: It belongs to an SCC that contains an affected source state. The reason is that the change in $V(s')$ of the affected source state s' can change $V(s)$ according to Equation 1 because states in an SCC form a cyclic subgraph.

[1] One can slightly optimize this function for UCT by only reinitializing $Q(s, a)$ and $n_{s,a}$ for all state-action pairs (s, a) with either (1) changes in $T(s, a, s')$ or $C(s, a, s')$ for some state s'; or (2) non-reusable successor states, that is, $T(s, a, s') > 0$ and $s'.reusable = false$ for some state s'.

[2] A similar proof applies for sampling algorithms like UCT.

Case 3: It belongs to an SCC that is upstream of an SCC c' that contains an affected source state. The reason is that the change in $V(s')$ of a state $s' \in c'$ can change $V(s'')$ of a state $s'' \in Pred(c')$ with transition function $T(s'', a, s') > 0$, which can then change the value function for all states in the SCC $Pred(c')$, and this effect propagates up to all SCCs upstream of SCC c'. ■

Lemma 2 *A state is non-reusable iff it is marked non-reusable.*

Proof Sketch: If a state is non-reusable, then it must fall into one of the following two cases according to Lemma 1:

Case 1: It belongs to an SCC that contains an affected source state. In this case, the SCC it belongs to is marked non-reusable (Line 20) and the state is marked the same (Lines 27-29).

Case 2: It belongs to an SCC that is upstream of an SCC that contains an affected source state. In this case, the SCC it belongs to is marked non-reusable (Lines 23-26 and that the SCC indices are in reverse topological sort order) and the state is marked the same (Lines 27-29). ■

Theorem 1 *The Incremental MDP Framework is sound.*

Proof Sketch: The framework correctly marks states as reusable or otherwise according to Lemma 2 and only reinitializes the state information of non-reusable states in Line 12. Thus, a sound MDP algorithm that can be bootstrapped to start with correct state information for some states remains sound when it uses this framework. ■

Property 1 *The time and space complexity of Lines 10-12 are both* $O(|\mathbf{S}|^2|\mathbf{A}|)$.

The time complexity of FIND-SCCS(), which is Tarjan's algorithm, is linear in the number of nodes $O(|\mathbf{S}|)$ and edges $O(|\mathbf{S}|^2|A|)$ in the graph since it runs a depth-first search over the transition graph.[3] The time complexity of FIND-REUSABLE-STATES() is dominated by Lines 23-24, which is linear in the number of SCCs $O(|\mathbf{S}|)$ and edges between SCCs $O(|\mathbf{S}|^2|\mathbf{A}|)$. The time complexity of UPDATE-INFORMATION() is $O(|\mathbf{S}|)$ for algorithms based on Bellman updates since it needs to update every non-reusable state and $O(|\mathbf{S}|^2|\mathbf{A}|)$ for algorithms based on sampling since it needs to update every non-reusable state-action pair. The space required by all three functions is dominated by the space required to store the MDP model M, which is $O(|\mathbf{S}|^2|\mathbf{A}|)$ for the transition and cost functions.

Property 2 *Assuming that every SCC has the same number of states, the estimated savings in runtime for using the Incremental MDP framework is* $|\hat{Y}| \cdot |Y| \cdot \frac{\tilde{T}}{|\tilde{Y}|^2}$ *for VI and* $|\hat{Y}| \cdot \frac{\tilde{T}}{|\tilde{Y}|}$ *for TVI, where* $|\hat{Y}|$ *is the number of reusable SCCs in the SCC transition tree,* $|Y|$ *is the number of SCCs in that tree,* $\tilde{T}$ *is runtime for solving the initial MDP and* $|\tilde{Y}|$ *is the number of SCCs in the SCC transition tree of the initial MDP.*

[3]There are $O(|\mathbf{S}|^2|\mathbf{A}|)$ edges because there are $O(|\mathbf{S}||\mathbf{A}|)$ hyper-edges, and each hyper-edge can connect up to $O(|\mathbf{S}|)$ nodes.

We now describe the high-level ideas to derive this property. Assume that states in every SCC need to be updated before the algorithm can converge. Then, the runtime T is $|Y| \cdot k \cdot t$ without our framework and is $(|Y| - |\hat{Y}|) \cdot k \cdot t$ with our framework, where k is the number of iterations necessary for one SCC to converge and t is the runtime per iteration. Also, due to the premise of the property, $t = O(|Y|)$ for VI and $t = O(1)$ for TVI. The number of iterations k is mostly unchanged if the problem does not change significantly. After solving the initial MDP, it can be estimated to be $O\big(\frac{\tilde{T}}{|\tilde{Y}|^2}\big)$ for VI and $O\big(\frac{\tilde{T}}{|\tilde{Y}|}\big)$ for TVI.

Experimental Results

We evaluate the Incremental MDP Framework with the 5 MDP algorithms described earlier. We run the algorithms on two sets of domains: (*i*) a multi-layer navigation domain similar to that in the literature (Dai et al. 2011), and (*ii*) the domains in the ICAPS 2011 International Probabilistic Planning Competition (IPPC).[4] We conducted our experiments on a dual-core 2.53 GHz machine with 4GB of RAM. All data points are averaged over 10 runs and runtimes are in milliseconds. We provide runtime results for when the algorithms use the zero heuristic as well as the deterministic heuristic. The runtimes do not include the pre-processing time to compute the deterministic heuristic.

Multi-Layer Navigation Domain

In this domain, an agent needs to get from one side of a river to the other. We model this problem as an $X \times Y$ grid, where states are cells in the grid and the number of rows Y in the grid is the width of the river. The agent can transition to any cell in its row or any of the 50 rows in front of it. We set the number of actions per state and the number of successors per action from 1 to 7 randomly and we set the transition and cost functions randomly. The agent has a sensor with range R. Thus, it can check for inconsistencies between its model and the environment for R rows ahead of it.

We set X to 50, Y to 500, and R to 50. The agent will continue to move according to its policy until it reaches its goal or detects an inconsistency and replans with our framework or from scratch. For each problem, we change the transition function of one random state. We set the coordinate (x, y) of this state by randomly choosing x such that that state is in the subgraph rooted at the start state and varying y from R to Y.

Tables 1(a) and 1(b) show the convergence runtimes when the algorithms use the zero and deterministic heuristics, respectively. Algorithms using our Incremental MDP framework have prefix "I" (which stands for "incremental") in their acronyms. We do not include the runtimes of UCT since it is an anytime algorithm that is designed without a test for convergence (Kolobov, Mausam, and Weld 2012). We show actual savings in runtime in parentheses and estimated savings (Property 2) in square brackets. The runtimes

[4]http://users.cecs.anu.edu.au/~ssanner/IPPC_2011

(a) with zero heuristic

y-coordinate	VI	I-VI	TVI	I-TVI	ILAO*	I-ILAO*	LRTDP	I-LRTDP
50	500	22 (478) [488]	35	20 (15) [34]	1565	19 (1546)	848	33 (815)
140	358	18 (340) [350]	29	17 (12) [27]	1196	16 (1180)	565	16 (548)
230	199	17 (182) [191]	24	14 (10) [23]	829	12 (816)	323	12 (311)
320	128	12 (116) [119]	16	9 (7) [15]	336	9 (327)	173	9 (164)
410	81	7 (74) [73]	9	6 (3) [8]	41	5 (36)	37	5 (32)
500	13	13 (0) [0]	3	3 (0) [0]	3	3 (0)	5	4 (1)

(b) with deterministic heuristic

y-coordinate	VI	I-VI	TVI	I-TVI	ILAO*	I-ILAO*	LRTDP	I-LRTDP
50	435	21 (414) [425]	34	19 (15) [33]	1548	18 (1530)	836	33 (803)
140	310	17 (293) [303]	29	16 (13) [28]	1156	16 (1141)	545	16 (529)
230	162	14 (149) [156]	22	12 (9) [20]	704	11 (693)	321	12 (310)
320	117	11 (106) [110]	17	9 (7) [15]	323	9 (314)	195	9 (186)
410	80	7 (73) [72]	10	6 (4) [8]	37	5 (32)	37	5 (32)
500	14	15 (-1) [0]	3	4 (0) [0]	2	3 (0)	5	4 (1)

Table 1: Multi-Layer Navigation Domain: Runtimes. *Actual savings are in parentheses and estimated savings are in square brackets.*

of all algorithms decrease with increasing y-coordinate because the problem size (= number of reachable states) decreases with increasing y-coordinate. The runtimes of all the algorithms are generally smaller if they use our framework. The savings in runtime decrease with decreasing problem size because the number of reusable states decreases with the problem size but the number of non-reusable states (states in the R rows ahead of the agent) remain the same. When there is only a small number of reusable states, the algorithms that use our framework converge slower due to additional overhead.

Next, we conducted an experiment to compare the quality of solutions found by LRTDP and UCT when they are both given the same amount of runtime to replan. Both algorithms are given about 30 seconds to find the first policy. We use the same parameters as earlier except for Y, which we set to 3000 so that the algorithms take longer to converge and their anytime behavior is more apparent, and R, which we set to 1000.

Table 2 shows the results. When given the same amount of time, (I-)LRTDP finds better solutions (= smaller costs) than (I-)UCT, consistent with recent results in the literature (Kolobov, Mausam, and Weld 2012). As expected, both LRTDP and UCT also finds better solutions when they use our framework to reuse information. This difference decreases as more time is given until both algorithms find the same plan when they both converge within the time limit.

Runtimes	LRTDP	I-LRTDP	UCT	I-UCT
256	2233	1719 (514)	2732	2075 (657)
512	2148	1744 (404)	2664	2050 (614)
1024	2074	1801 (273)	2579	2014 (565)
2048	2038	1716 (321)	2496	1987 (509)
4096	1999	1500 (499)	2411	1964 (448)
8192	1878	1507 (371)	2275	1927 (348)
16384	1539	1470 (69)	2139	1877 (262)
32768	1425	1425 (0)	1994	1827 (167)

Table 2: Multi-Layer Navigation Domain: Expected Costs. *Cost differences are in parentheses.*

ICAPS 2011 IPPC Domains

All eight IPPC domains are represented as finite-horizon MDPs. Out of these eight domains, four domains (*crossing traffic*, *navigation*, *reconnaissance*, and *skill teaching*) can be represented as general SSP-MDPs as it is relatively straightforward to define goal states in these domains. We thus use these domains in our experiments. We do not use the remaining four domains as they are repeated scheduling problems without clearly defined goal states. For the *crossing traffic* and *skill teaching* domains, we report the results of two largest instance that fit in memory. For the *navigation* domain, all the instances were too small and could not sufficiently illustrate the difference in runtimes of the various algorithms, and for the *reconnaissance* domain, all the instances were too large and could not fit in memory. As such, we created two larger instances for the *navigation* domain and two smaller instances for the *reconnaissance* domain using the same domain logic and report the results of those instances.

In these experiments, we change the transition function of one random state, which the agent is able to detect only if it is at the predecessor state of this random state. Additionally, we also run a pre-processing step to eliminate provably suboptimal actions like reversible actions to increase the number of SCCs (Dai et al. 2011).

Tables 3(a) and 3(b) show the convergence runtimes when the algorithms use the zero and deterministic heuristics, respectively. Both VI and TVI converged faster in all domains and both heuristics when they use our Incremental MDP framework. Additionally, the estimated savings are more

(a) with zero heuristic

Domains	VI	I-VI			TVI	I-TVI			ILAO*	I-ILAO*		LRTDP	I-LRTDP	
crossing traffic	611	171	(440)	[610]	192	178	(14)	[191]	550	184	(366)	174	142	(32)
navigation	10833	22	(10811)	[10821]	330	20	(310)	[329]	28	23	(5)	42	23	(19)
reconnaissance	11	3	(8)	[10]	7	5	(2)	[6]	6	4	(2)	13	4	(9)
skill teaching	88	6	(82)	[71]	44	6	(38)	[34]	87	6	(81)	239	14	(225)

(b) with deterministic heuristic

Domains	VI	I-VI			TVI	I-TVI			ILAO*	I-ILAO*		LRTDP	I-LRTDP	
crossing traffic	593	168	(425)	[592]	187	176	(11)	[186]	511	176	(335)	126	125	(1)
navigation	1052	19	(1033)	[1051]	177	19	(158)	[176]	1	19	(-18)	1	20	(-19)
reconnaissance	6	2	(4)	[5]	4	3	(1)	[3]	2	3	(-1)	2	3	(-1)
skill teaching	97	7	(90)	[77]	50	8	(42)	[39]	87	8	(79)	237	18	(219)

Table 3: ICAPS 2011 IPPC Domains: Runtimes. *Actual savings are in parentheses and estimated savings are in square brackets.*

Domains	VI	I-VI			TVI	I-TVI		
crossing traffic	38	20	(18)	[37]	23	21	(2)	[22]
navigation	35	19	(16)	[34]	26	19	(7)	[25]
reconnaissance	3	3	(0)	[2]	5	4	(1)	[4]
skill teaching	6	4	(2)	[5]	7	4	(3)	[5]

Table 4: ICAPS 2011 IPPC Domains: Runtimes with Old-Value Heuristic. *Actual savings are in parentheses and estimated savings are in square brackets.*

Weights	ILAO*	I-ILAO*	LRTDP	I-LRTDP
0.00	28	23 (5)	42	23 (19)
0.25	26	23 (3)	43	24 (19)
0.50	21	24 (-3)	35	25 (10)
0.75	11	22 (-11)	16	23 (-7)
1.00	1	19 (-18)	1	20 (-19)

Table 5: ICAPS 2011 IPPC's *Navigation* Domain: Runtimes. *Actual savings are in parentheses.*

accurate in the multi-layer navigation domain than in these IPPC domains. The reason is that Property 2 assumes that every SCC has the same number of states, and the variance in the number of states in each SCC is smaller in the former domain than in the latter domains.

Finally, instead of reinitializing the value function $V(s)$ of non-reusable states s in VI and TVI to their heuristic values, one can keep their old values as they will not affect the correctness of both algorithms. Table 4 shows the results for this "old-value heuristic". As this heuristic is more informed than the precomputing heuristic, the runtimes of all the algorithms are smaller and, consequently, the runtime savings are also smaller. The results are similar in the multi-layer navigation domain.

Tables 3(a) and 3(b) also show that when ILAO* and LRTDP use the zero heuristic, they converged faster in all domains when they use our Incremental MDP framework. When they use the deterministic framework, they converged faster only in the *crossing traffic* and *skill teaching* domains. Based on this observation, we wanted to investigate the impact of the informedness of heuristic values on the runtime savings of our approach. We thus re-ran the experiments on the *navigation* domain using different weights on the deterministic heuristics. Table 5 shows the runtime results. When the heuristics are sufficiently informed, it is faster to not use the Incremental MDP framework. The reason is that the overhead incurred by the framework is larger than the savings gained by it.

Discussions and Conclusions

In this paper, we investigate the feasibility of a new approach to solve Uncertain MDPs by repeatedly solving the current MDP when a change is detected. Unlike previous approaches, this approach does not require a model of the uncertainty in the parameters of the MDP model. Using this approach, we introduced a general framework that allows off-the-shelf MDP algorithms to reuse state information that are unaffected by the change in the MDP. Our experimental results show that our approach is typically faster than replanning from scratch and finds better solutions when given a fixed runtime. The savings in runtime decreases with the increase in informedness of heuristics used. Thus, it is well-suited in domains where good heuristics are difficult to compute, which is the case in many complex application domains. Future work includes characterizing changes to the problem that violate or retain the SSP-MDP property as well as comparing this approach with previous proactive approaches.

Acknowledgment

This research is partially supported by the National Science Foundation under grant number HRD-1345232. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

Ahmed, A.; Varakantham, P.; Adulyasak, Y.; and Jaillet, P. 2013. Regret based robust solutions for uncertain Markov decision processes. In *Advances in Neural Information Processing Systems (NIPS)*, 881–889.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bertsekas, D. 2000. *Dynamic Programming and Optimal Control*. Athena Scientific.

Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1233–1238.

Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 12–21.

Dai, P.; Mausam; Weld, D.; and Goldsmith, J. 2011. Topological value iteration algorithms. *Journal of Artificial Intelligence* 42(1):181–209.

Delage, E., and Mannor, S. 2010. Percentile optimization for Markov decision processes with parameter uncertainty. *Operations Research* 58(1):203–213.

Givan, R.; Leach, S.; and Dean, T. 2000. Bounded-parameter Markov decision processes. *Artificial Intelligence* 122(1–2):71–109.

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

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC4(2):100–107.

Iyengar, G. 2005. Robust dynamic programming. *Mathematics of Operations Research* 30:257–280.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning (ECML)*, 282–293.

Koenig, S., and Likhachev, M. 2002. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 476–483.

Koenig, S.; Likhachev, M.; Liu, Y.; and Furcy, D. 2004. Incremental heuristic search in AI. *AI Magazine* 25(2):99–112.

Kolobov, A.; Mausam; and Weld, D. 2012. LRTDP vs. UCT for online probabilistic planning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1786–1792.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS)*.

Mannor, S.; Simester, D.; Sun, P.; and Tsitsiklis, J. 2007. Bias and variance approximation in value function estimates. *Management Science* 53:308–322.

Mannor, S.; Mebel, O.; and Xu, H. 2012. Lightning does not strike twice: Robust MDPs with coupled uncertainty. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Nilim, A., and Ghaoui, L. E. 2005. Robust Markov decision processes with uncertain transition matrices. *Operations Research* 53(5):780–798.

Regan, K., and Boutilier, C. 2009. Regret-based reward elicitation for Markov decision processes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 444–451.

Regan, K., and Boutilier, C. 2011. Robust online optimization of reward-uncertain MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2165–2171.

Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1652–1659.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 469–476.

Sun, X.; Yeoh, W.; and Koenig, S. 2009. Efficient incremental search for moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 615–620.

Sun, X.; Yeoh, W.; and Koenig, S. 2010a. Generalized Fringe-Retriving A*: Faster moving target search on state lattices. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 1081–1087.

Sun, X.; Yeoh, W.; and Koenig, S. 2010b. Moving Target D* Lite. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 67–74.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2):146–160.

Xu, H., and Mannor, S. 2012. Distributionally robust Markov decision processes. *Mathematics of Operations Research* 37(2):288–300.