Solving Concurrent Markov Decision Processes

Mausam and Daniel S. Weld
Dept of Computer Science and Engineering
University of Washington
Seattle, WA-98195
{mausam,weld}@cs.washington.edu

Content areas: Markov Decision Processes, Planning
Technical Report UW-CSE-TR-04-04-03

Abstract

Typically, Markov decision problems (MDPs) assume a single action is executed per decision epoch, but in the real world one may frequently execute certain actions in parallel. This paper explores *concurrent MDPs*, MDPs which allow multiple non-conflicting actions to be executed simultaneously, and presents two new algorithms. Our first approach exploits two provably sound pruning rules, and thus guarantees solution optimality. Our second technique is a fast, sampling-based algorithm, which produces close-to-optimal solutions extremely quickly. Experiments show that our approaches outperform the existing algorithms producing up to two orders of magnitude speedup.

1 Introduction

Recent progress achieved by planning researchers has yielded new algorithms that relax, individually, many of the classical assumptions. However, in order to apply automated planning to many real-world domains we must eliminate larger groups of the assumptions in concert. For example, Bresina *et al.* [6] note that optimal control for a NASA Mars rover requires reasoning about uncertain, concurrent, durative actions and a mixture of discrete and metric fluents. While today's planners can handle large problems with *deterministic* concurrent durative actions, and semi-MDPs provide a clear framework for durative actions in the face of uncertainty, few researchers have considered concurrent, uncertain actions — the focus of this paper.

For example, a Mars rover has the goal of gathering data from different locations with various instruments (color and infrared cameras, microscopic imager, Mossbauer spectrometers *etc.*) and transmitting this data back to Earth. Concurrent actions are essential since instruments can be turned on, warmed up and calibrated while the rover is moving, using other instruments or transmitting data. Similarly, uncertainty must be explicitly confronted as the rover's movement, arm control and other actions cannot be accurately predicted.

We adopt the framework of *Markov decision processes* (MDPs) and extend it to allow multiple actions per decision epoch. In the traditional case of a single action per decision epoch, state-space heuristic search and dynamic programming have proven quite effective. However, allowing multiple concurrent actions at a time point will inflict an exponential blowup on all of these techniques.

In this paper we investigate techniques to counter this combinatorial explosion. Specifically, we extend the technique of *real-time dynamic programming* (RTDP) [1, 4] to handle concurrency, making the following contributions:

- We empirically illustrate the exponential blowup suffered by the existing MDP algorithms.
- We describe two pruning strategies (*combo-elimination* and *combo-skipping*), prove that they preserve completeness, and evaluate their performance.
- We describe a novel technique, *combo-sampling*, that produces a speedup of an order of magnitude. Although this technique sacrifices solution optimality, we show that for a wide range of problems, combo-sampling produces solutions that are quite close to optimal.

2 Background

Planning problems under probabilistic uncertainty are often modeled using Markov Decision Processes (MDPs). Different research communities have looked at slightly different formulations of MDPs. These versions typically differ in objective functions (maximising reward *vs.* minimising cost), horizons (finite, infinite, indefinite) and action representations (DBN *vs.* parametrised action schemata). All these formulations are very similar in nature, and so are the algorithms to solve them. Though, the methods proposed in the paper are applicable to all the variants of these models, for clarity of explanation we assume a particular formulation of an MDP as follows.

Following Bonet and Geffner [4], we define a *Markov decision process* as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}, \mathcal{G}, s_0, \gamma \rangle$ in which

- $\mathcal{S}$ is a finite set of discrete states.
- $\mathcal{A}$ is a finite set of actions. An applicability function, $Ap : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$, denotes the set of actions that can be applied in a given state ($\mathcal{P}$ represents the power set).
- $\mathcal{P}r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function. We write $\mathcal{P}r(s'|s, a)$ to denote the probability of arriving at state s' after executing action a in state s.
- $\mathcal{C} : \mathcal{A} \rightarrow \Re^+$ is the cost model[1].
- $\mathcal{G} \subseteq \mathcal{S}$ is a set of absorbing goal states.
- s_0 is a start state.
- $\gamma \in [0, 1]$ is the discount factor. If $\gamma = 1$ our problem is known as the *stochastic shortest path problem* [2].

We assume full observability, and we seek to find an optimal, stationary policy — ***i.e.***, a function π: $\mathcal{S} \rightarrow \mathcal{A}$ that minimises the expected discounted cost (over an infinite horizon) incurred to reach a goal state. Note that any *value function*, J: $\mathcal{S} \rightarrow \Re$, mapping states to the expected cost of reaching a goal state defines a policy as follows:

$$\pi_J(s) = \operatorname*{argmin}_{a \in Ap(s)} \left\{ \mathcal{C}(a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}r(s'|s, a) J(s') \right\}$$

The *optimal* policy derives from a value function, J^*, which satisfies the following pair of *Bellman equations*.

$$\begin{aligned} J^*(s) &= 0, \text{ if } s \in \mathcal{G} \text{ else} \\ J^*(s) &= \min_{a \in Ap(s)} \left\{ \mathcal{C}(a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}r(s'|s, a) J^*(s') \right\} \end{aligned} \tag{1}$$

For example, Figure 1 defines a simple MDP where four state variables ($x_1, \ldots, x_4$) need to be set using toggle actions. Some of the actions, ***e.g.***, toggle-x_3 are probabilistic.

[1]Indeed, all our techniques except Theorem 2 allow costs to be conditioned on states as well as actions.

State variables : $x_1, x_2, x_3, x_4, p_{12}$

Action	Precondition	Effect	Probability
toggle-x_1	$\neg p_{12}$	$x_1 \leftarrow \neg x_1$	1
toggle-x_2	p_{12}	$x_2 \leftarrow \neg x_2$	1
toggle-x_3	true	$x_3 \leftarrow \neg x_3$	0.9
		no change	0.1
toggle-x_4	true	$x_4 \leftarrow \neg x_4$	0.9
		no change	0.1
toggle-p_{12}	true	$p_{12} \leftarrow \neg p_{12}$	1

Goal : $x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1$

Figure 1: Probabilistic STRIPS definition of a simple MDP with potential parallelism

Various algorithms have been developed to solve MDPs. *Value iteration* is a dynamic programming approach in which the optimal value function (the solution to equations 1) is calculated as the limit of a series of approximations, each considering increasingly long action sequences. If $J_n(s)$ is the value of state s in iteration n, then the value of state s in the next iteration is calculated with a process called a *Bellman backup* as follows:

$$J_{n+1}(s) = \min_{a \in Ap(s)} \left\{ \mathcal{C}(a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}r(s'|s,a) J_n(s') \right\}$$

Value iteration terminates when $\forall s \in \mathcal{S},\ |J_n(s) - J_{n-1}(s)| \leq \epsilon$, and this termination is guaranteed for $\epsilon > 0$. Furthermore, the sequence of $\{J_i\}$ is guaranteed to converge to the optimal value function, J^*, regardless of the initial values. Unfortunately, value iteration tends to be quite slow, since it explicitly updates every state, and $|\mathcal{S}|$ is exponential in the number of domain features. One optimization restricts search to the part of state space reachable from the initial state s_0. Two algorithms exploiting this *reachability analysis* are LAO* [10] and our focus: RTDP [1].

RTDP, conceptually, is a lazy version of value iteration in which the states get updated in proportion to the frequency with which they are visited by the repeated executions of the greedy policy. Specifically, RTDP is an anytime algorithm that simulates the greedy policy along a single trace execution, and updates the values of the states it visits using Bellman backups. An RTDP *trial* is a path starting from s_0 and ending when a goal is reached or the number of updates exceeds a threshold. RTDP repeats these trials until convergence. Note that common states are updated frequently, while RTDP wastes no time on states that are unreachable, given the current policy. RTDP's strength is its ability to quickly produce a relatively good policy; however, complete convergence (at every state) is slow because less likely (but potentially important) states get updated infrequently. Furthermore, RTDP is not guaranteed to terminate. *Labeled RTDP* fixes these problems with a clever labeling scheme that focusses attention on states where the value function has not yet converged [4]. Labeled RTDP is guaranteed to terminate, and is guaranteed to converge to the optimal value function (for states reachable using the optimal policy) if the initial value function is admissible.

3 Concurrent Markov Decision Processes

Extending traditional MDPs to *concurrent MDPs*, *i.e.* allowing multiple parallel actions, each of unit duration, requires several changes. Clearly, certain actions can't be executed in parallel; so we adopt the classical planning notion of mutual exclusion [3] and apply it to a *factored* action representation: *probabilistic STRIPS* [5]. Two actions are *mutex* (may not be executed concurrently) if in any state 1) they have inconsistent preconditions, 2) they have conflicting effects, or 3) the precondition of one conflicts with the (possibly probabilistic) effect of the other. Thus, non-mutex actions don't interact — the effects of executing the sequence $a_1; a_2$ equals those for $a_2; a_1$.

Example: Continuing with Figure 1, toggle-x_1, toggle-x_3 and toggle-x_4 can execute in parallel but toggle-x_1 and toggle-x_2 are mutex as they have conflicting preconditions. Similarly, toggle-x_1 and toggle-p_{12} are mutex as the effect of toggle-p_{12} interferes with the precondition of toggle-x_1.

3.1 Cost model

An *action combination*, A, is a set of one or more actions to be executed in parallel. The cost model $\mathcal{C}$ is now a function, $\mathcal{C} : \mathcal{P}(\mathcal{A}) \rightarrow \Re^+$, *i.e.* the domain is the *power-set* of actions. Note that unless there exists a combination A, such that $\mathcal{C}(A) < \sum_{a \in A} \mathcal{C}(\{a\})$, the optimal policy from the single-action MDP would be optimal for the concurrent case as well. However, we believe that in many domains most combinations do obey the inequality. Indeed, the inequality always holds when the cost of a combination includes both *resource* and *time* components. Here, one can define the cost model to be comprised of two parts:

- t : Time taken to complete the action.
- r : Amount of resources used for the action.

Assuming additivity, we can think of cost of an action $\mathcal{C}(a) = t(a) + r(a)$, to be sum of its time and resource usage. Hence, the cost model for a combination of actions in terms of these components would be defined as:

$$\mathcal{C}(\{a_1, a_2, ..., a_k\}) = \sum_{i=1}^{k} r(a_i) + \max_{i=1..k}\{t(a_i)\}$$

For example, a Mars rover might incur lower cost when it preheats an instrument while changing locations than if it executes the actions sequentially, because the makespan is reduced while the energy consumed does not change.

3.2 Applicability Function

The applicability function, $Ap(s)$, for concurrent MDPs now has range $\mathcal{P}(\mathcal{P}(\mathcal{A}))$; it is redefined in terms of our original definition, now denoted Ap_1. $Ap(s) = \{A \subseteq \mathcal{A} | \forall a, a' \in A,\ a, a' \in Ap_1(s) \wedge\ \neg \mathrm{mutex(a, a')}\}$

3.3 Transition Function

Let $A = \{a_1, a_2, \ldots, a_k\}$ be an action combination applicable in s. Since the actions don't interact, the transition function may be calculated as follows:

$$\begin{aligned}\mathcal{P}r(s'|s, A) = & \sum_{s_1, s_2, \ldots s_k \in \mathcal{S}} \!\!\!\ldots \sum \\ & \mathcal{P}r(s_1|s, a_1)\mathcal{P}r(s_2|s_1, a_2) \ldots \mathcal{P}r(s'|s_k, a_k)\end{aligned}$$

3.4 Bellman equations

Finally, instead of equations (1), the following set of equations represents the solution to a concurrent MDP:

$$\begin{aligned}&J^*(s) = 0, \text{ if } s \in \mathcal{G} \text{ else}\\ &J^*(s) = \min_{A \in Ap(s)} \left\{ \mathcal{C}(A) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}r(s'|s, A) J^*(s') \right\}\end{aligned} \tag{2}$$

These equations are the same as in a traditional MDP, except that instead of considering single actions for backup in a state, we need to consider all applicable action combinations. Thus, only this small change must be made to traditional algorithms (*e.g.*, value iteration, LAO*, Labeled RTDP). However since the number of action combinations is exponential in $|\mathcal{A}|$, efficiently solving a concurrent MDP requires new techniques. Unfortunately, there is no easy structure to exploit, since an optimal action for a state from a classical MDP solution may not even appear in the optimal action combination for a concurrent MDP.

Lemma 1 *All actions in optimal combination for Concurrent MDP may individually be sub-optimal for the Classical MDP.*

In the next section, we describe two provably-sound pruning techniques that speed policy construction; then in Section 5, we present fast sampling methods which generate near-optimal policies.

4 Pruned RTDP

Recall that during a trial, labeled RTDP performs Bellman backups in order to calculate the values of applicable actions (or in our case, action combinations) and then chooses the best action (combination); we now describe two pruning techniques that reduce the number of backups to be computed. Also, let $Q(s,A)$ be the expected cost incurred by executing action combination A in state s and then following the greedy policy, *i.e.*

$$Q_n(s,A) = \mathcal{C}(A) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}r(s'|s,A) J_{n-1}(s')$$

A Bellman update can thus be rewritten as:

$$J_n(s) = \min_{A \in Ap(s)} Q_n(s,A)$$

4.1 Combo Skipping

Since the number of applicable action combinations can be exponential, we'd like to prune suboptimal combinations. The following theorem imposes a lower bound on $Q(s,A)$ in terms of the costs and the Q-values of single actions.

Theorem 2 *Let $A = \{a_1, a_2, \ldots, a_k\}$ be an action combination which is applicable in state s.*

$$Q(s,A) \geq \gamma^{1-k} Q(s,\{a_1\}) + \mathcal{C}(A) - \left(\sum_{i=1}^{k} \gamma^{i-k} \mathcal{C}(\{a_i\}) \right)$$

Proof:

$$\begin{aligned} Q_n(s,A) &= \mathcal{C}(A) + \gamma \sum_{s'} \Pr(s'|s,A) J_{n-1}(s') \\ \Rightarrow \sum_{s'} \Pr(s'|s,A) J_{n-1}(s') &= \frac{1}{\gamma}(Q_n(s,A) - \mathcal{C}(A)) \end{aligned}$$

$$\begin{aligned}
Q_n(s,\{a_1\}) &= \mathcal{C}(\{a_1\}) + \gamma \sum_{s''} \Pr(s''|s,a_1) J_{n-1}(s'') \\
&\leq \mathcal{C}(\{a_1\}) + \gamma \sum_{s''} \Pr(s''|s,a_1) \left[\mathcal{C}(\{a_2\}) + \gamma \sum_{s'''} \Pr(s'''|s'',a_2) J_{n-2}(s''') \right] \\
&= \mathcal{C}(\{a_1\}) + \gamma \mathcal{C}(\{a_2\}) + \gamma^2 \sum_{s'''} \Pr(s'''|s,\{a_1,a_2\}) J_{n-2}(s''') \\
&\leq \sum_{i=1}^{k} \gamma^{i-1} \mathcal{C}(\{a_i\}) + \gamma^k \sum_{s'} \Pr(s'|s,A) J_{n-k}(s') \\
&= \sum_{i=1}^{k} \gamma^{i-1} \mathcal{C}(\{a_i\}) + \gamma^{k-1} [Q_{n-k+1}(s,A) - \mathcal{C}(A)]
\end{aligned}$$

$$\begin{aligned}
Q_n(s,A) &\geq \gamma^{1-k} Q_{n+k-1}(s,\{a_1\}) + \mathcal{C}(A) - \gamma^{1-k} \left(\sum_{i=1}^{k} \gamma^{i-1} \mathcal{C}(\{a_i\}) \right) \\
&\geq \gamma^{1-k} Q_n(s,\{a_1\}) + \mathcal{C}(A) - \gamma^{1-k} \left(\sum_{i=1}^{k} \gamma^{i-1} \mathcal{C}(\{a_i\}) \right)
\end{aligned}$$

Corollary 3 *Let $\lceil J_n(s) \rceil$ be an upper bound of $J_n(s)$. If*

$$\lceil J_n(s) \rceil < \gamma^{1-k} Q_n(s,\{a_1\}) + \mathcal{C}(A) - \left(\sum_{i=1}^{k} \gamma^{i-k} \mathcal{C}(\{a_i\}) \right)$$

then, A cannot be optimal for state s in this iteration.

Corollary 3 justifies a pruning rule, *combo-skipping*, that preserves optimality in any iteration algorithm that maintains value function monotonicity. This is powerful because all Bellman-backup based algorithms preserve monotonicity when started with an admissible value function. To apply combo-skipping, one must compute all the $Q(s,\{a\})$ values for single actions a that are applicable in s; it is useful to precompute the summation of discounted costs, *i.e.* $\left(\sum_{i=1}^{k} \gamma^{i-k} \mathcal{C}(\{a_i\}) \right)$, for all possible combinations. In the undiscounted case, this computation reduces to the simple sum of costs. To calculate $\lceil J_n(s) \rceil$ one may use the optimal combination for state s in the previous iteration (A_{opt}) and compute $Q_n(s, A_{opt})$. This value gives an upper bound on the value $J_n(s)$.

Theorem 2 and Corollary 3 are valid for any ordering of a_i's. But in order to skip the most combinations, we must maximise the right-hand side. In practice, the following heuristic suffices: choose a_1 to be the a_i with maximal $Q(s, a_i)$ and order other actions in order of increasing cost.

Example: In Figure 1, let γ=1. Let a single action incur unit cost, and let the cost of an action combination be: $\mathcal{C}(A)$ = 0.5 + 0.5$|A|$. Let state s = (1,1,0,0,1) represent the ordered values x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, and p_{12} = 1. Suppose, after the n^{th} iteration, the value function assigns the values: $J_n(s)$ = 1, $J_n(s_1$=(1,0,0,0,1)) = 2, $J_n(s_2$=(1,1,1,0,1)) = 1, $J_n(s_3$=(1,1,0,1,1)) = 1. Let A_{opt} for state s be {toggle-x_3, toggle-x_4}. Now, $Q_{n+1}(s,$ {toggle-x_2}) = $\mathcal{C}$(toggle-x_2) + $J_n(s_1)$ = 3 and $Q_{n+1}(s, A_{opt})$ = $\mathcal{C}(A_{opt})$ + 0.81×0 + 0.09×$J_n(s_2)$ + 0.09×$J_n(s_3)$ + 0.01×$J_n(s)$ = 1.69. So now we can apply Corollary 3 to skip combination {toggle-x_2, toggle-x_3} in this iteration, since using toggle-x_2 as a_1, we have $\lceil J_{n+1}(s) \rceil$ = $Q_{n+1}(s, A_{opt})$ = 1.69 $\leq$ 3 + 1.5 - 2 = 2.5.

Experiments in Section 6 show that combo-skipping yields considerable savings. Unfortunately, combo-skipping has a weakness — it prunes a combination for only a *single iteration*. In contrast, our second rule, *combo-elimination*, prunes irrelevant combinations altogether.

4.2 Combo Elimination

We adapt the action elimination theorem from traditional MDPs [2] to prove a similar theorem for concurrent MDPs.

Theorem 4 *Let A be an action combination which is applicable in state s. Let $\lfloor Q^*(s, A) \rfloor$ denote a lower bound of $Q^*(s, A)$. If $\lfloor Q^*(s, A) \rfloor > \lceil J^*(s) \rceil$ then A is never the optimal combination for state s.*

In order to apply the theorem for pruning, one must be able to evaluate the upper and lower bounds. By using an admissible value function when starting RTDP search (or in value iteration, LAO* *etc.*), the current value $J_n(s)$ is guaranteed to be a lower bound of the optimal cost; thus, $Q_n(s, A)$ will also be a lower bound of $Q^*(s, A)$. Thus, it is easy to compute the left hand side of the inequality. To calculate an upper bound of the optimal $J^*(s)$, one may solve the MDP (*e.g.*, using labeled RTDP) while forbidding concurrency. This is much faster than solving the concurrent MDP, and yields an upper bound on cost, because forbidding concurrency restricts the policy to use a strict subset of legal action combinations.

Example: Continuing with the previous example, let A={toggle-x_2} then $Q_{n+1}(s, A)$ = $\mathcal{C}(A)$ + $J_n(s_1)$ = 3 and $\lceil J^*(s) \rceil$ = 2.222 (from solving MDP forbidding concurrency). As 3 > 2.222, A can be eliminated for state s in all remaining iterations.

Used in this fashion, combo-elimination requires the additional overhead of optimally solving the single-action MDP. Since algorithms like RTDP exploit state-space reachability to limit computation to relevant states, we do this computation incrementally, as new states are visited by our algorithm.

Combo-elimination also requires computing the current value of $Q(s, A)$ (for the lower bound of $Q^*(s, A)$); this differs from combo-skipping which avoids this computation. However, once combo-elimination prunes a combination, it never needs to be reconsidered. Thus, there is a tradeoff: should one perform an expensive computation, hoping for long-term pruning, or try a cheaper pruning rule with fewer benefits? Since Q-value computation is the costly step, we adopt the following heuristic: "First, try combo-skipping; if it fails to prune the combination, attempt combo-elimination". We also tried implementing some other heuristics, such as: 1) If some combination is being skipped repeatedly, then try to prune it altogether with combo-elimination. 2) In every state, try combo-elimination with probability p. Space precludes presenting our experimental results, but neither alternative performed significantly better, so we kept our original (lower overhead) heuristic.

Since combo-skipping does not change any step of labeled RTDP and combo-elimination removes provably sub-optimal combinations, *pruned* labeled RTDP maintains convergence, termination, optimality and efficiency, when used with an admissible heuristic.

5 Sampled RTDP

Since the fundamental challenge posed by concurrent MDPs is the explosion of action combinations, sampling is a promising method to reduce the number of Bellman backups required per state. We describe a variant of RTDP, called *sampled RTDP*, which performs backups on a random set of action combinations[2], choosing from a distribution that favors "likely combinations." We generate our distribution by: 1) using combinations that were previously discovered to have low Q-values (recorded by *memoizing* the best combinations per state, after each iteration); 2) calculating the Q-values of all applicable single actions (using current value function) and then biasing the sampling of combinations to choose the ones that contain actions with low Q-values.

This approach exposes an exploration / exploitation trade-off. Exploration, here, refers to testing a wide range of action combinations to improve understanding of their relative merit. Exploitation, on the other hand, advocates performing

[2]A similar action sampling approach was also used in the context of space shuttle scheduling to reduce the number of actions for value function computation [17].

backups on the combinations that have previously been shown to be the best. We manage the tradeoff by carefully maintaining the distribution over combinations. First, we only memoize best combinations per state; these are always backed-up number of combinations memoized. Other combinations are constructed by an incremental probabilistic process, which builds a combination by first randomly choosing an initial action (weighted by it's individual Q-value), then deciding whether to add a non-mutex action or stop growing the combination.

5.1 Termination and Optimality

Since the system doesn't consider every possible action combination, sampled RTDP is not guaranteed to choose the best combination to execute at each state. As a result, even when started with an admissible heuristic, the algorithm may assign $J_n(s)$ a cost that is greater than the optimal $J^*(s)$ — *i.e.*, the $J_n(s)$ values are no longer admissible. If a better combination is chosen in a subsequent iteration, $J_{n+1}(s)$ might be set a lower value than $J_n(s)$, thus sampled RTDP is not *monotonic*. This is unfortunate, since admissibility and monotonicity are important properties required for termination[3] and optimality in labeled RTDP; indeed, sampled RTDP loses these important theoretical properties. The good news is that it is extremely useful in practice. In our experiments, sampled RTDP usually terminates quickly, and returns values that are extremely close to the optimal.

5.2 Improving Solution Quality

We have investigated several heuristics in order to improve the quality of the solutions found by sampled RTDP.

- **Heuristic 1:** Whenever sampled RTDP asserts convergence of a state, do not immediately label it as converged (which would preclude further exploration [4]); instead first run a complete backup phase, using all the admissible combinations, to rule out any easy-to-detect inconsistencies.

- **Heuristic 2:** Run sampled RTDP to completion, and use the value function it produces, $J^s()$, as the initial heuristic estimate, $J_0()$, for a subsequent run of pruned RTDP. Usually, such a heuristic, though inadmissible, is highly informative. Hence, pruned RTDP terminates quite quickly.

- **Heuristic 3:** Run sampled RTDP before pruned RTDP, as in Heuristic 2, except instead of using the $J^s()$ value function directly as an initial esti-

[3]To ensure termination we implemented the policy: *if number of trials exceeds a threshold, force monotonicity on value function.* This will achieve termination but will reduce quality of solution.

mate, scale linearly downward — *i.e.*, use $J_0() := cJ^s()$ for some constant $c \in (0, 1)$. Hopefully, the estimate will be admissible (though there is no guarantee). In our experience, $c = 0.9$ suffices, and the run of pruned RTDP yields the optimal policy very quickly.

Experiments showed that Heuristic 1 returns a value function that is close to optimal. Adding Heuristic 2 improves this value moderately, and Heuristic 3 invariably returns the optimal solution.

6 Experiments

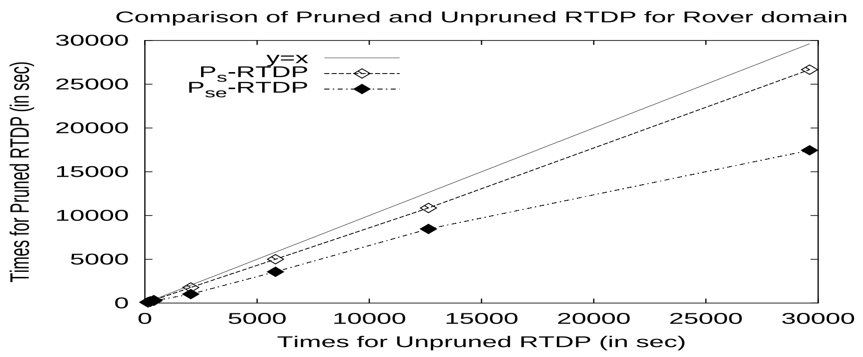
We tested our algorithms on problems in three domains. The first domain was a probabilistic variant of NASA Rover domain from the 2002 AIPS Planning Competition, in which there are multiple objects to be photographed and various rocks to be tested with resulting data communicated back to the base station. Cameras need to be focussed, and arms need to be positioned before usage. Since the rover has multiple arms and multiple cameras, the domain is highly parallel. The cost function includes both resource and time components, so executing multiple actions in parallel is cheaper than executing them sequentially[4]. We generated problems with 20-30 state variables having up to 81,000 reachable states and average number of applicable combinations per state ($Avg(Ap(s))$) up to 2735.

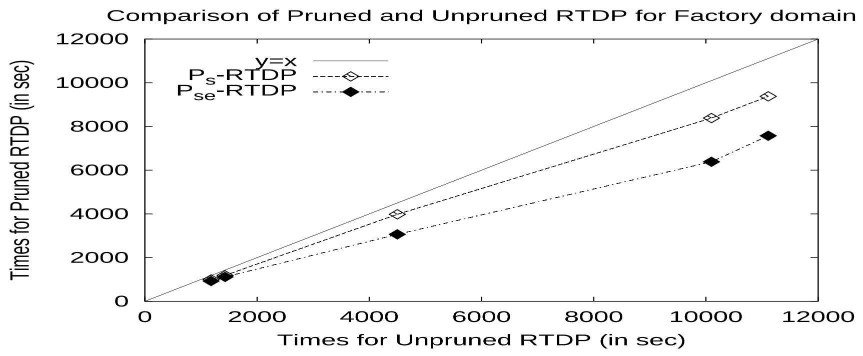
We also tested on a probabilistic version of a factory domain with multiple subtasks (*e.g.*, roll, shape, paint, polish *etc.*), which need to be performed on different objects using different machines. Machines can perform in parallel, but not all are capable of every task. We tested on problems with 26-28 state variables and around 32000 reachable states. $Avg(Ap(s))$ ranged between 170 and 2640.

Finally, we tested on an artificial domain similar to Figure 1 but much more complex. In this domain, some Boolean variables need to be toggled; however, toggling is probabilistic in nature. Moreover, certain pairs of actions have conflicting preconditions and thus, by varying the number of mutex actions we may control the domain's degree of parallelism. All the problems in this domain had 19 state variables and about 32000 reachable states, with $Avg(Ap(s))$ between 1024 and 12287.

We used Labeled RTDP, as implemented in GPT, as the base MDP solver. We implemented various algorithms, unpruned RTDP (U-RTDP), pruned RTDP using only combo skipping (P_s-RTDP), pruned RTDP using both combo skipping and combo elimination (P_{se}-RTDP), sampled RTDP using Heuristic 1 (S-RTDP) and

[4]For details on the domain, refer to `http://www.cs. washington.edu/ai/concurrent/NasaRover.pddl`





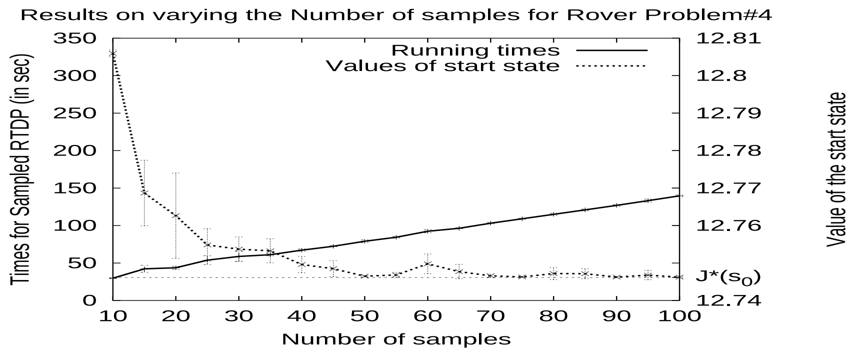
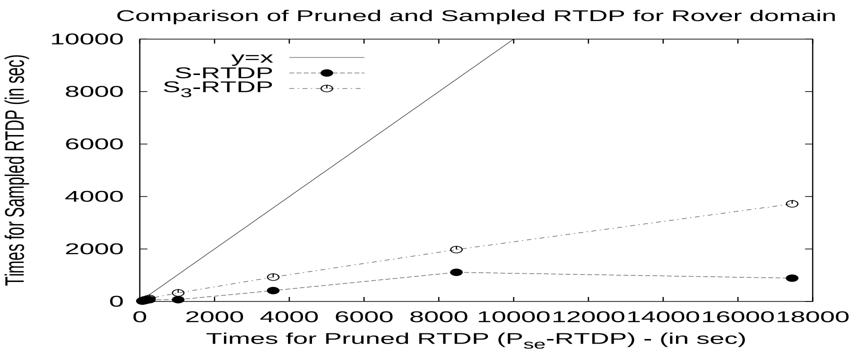
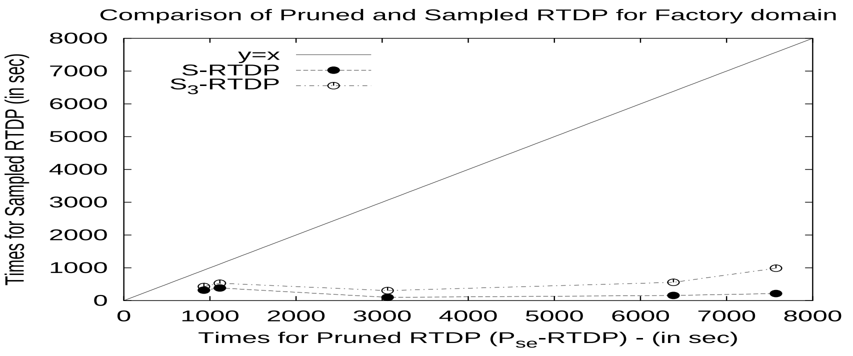


Figure 2: (a,b): Pruned vs. Unpruned RTDP for Rover and Factory domains respectively. Pruning non-optimal combinations achieves significant speedups on larger problems. (c) : Variation of quality of solution and efficiency of algorithm (with 95% confidence intervals) with the number of samples in Sampled RTDP for one particular problem from the Rover domain. As number of samples increase, the quality of solution approaches optimal and time still remains better than P_{se}-RTDP (which takes 259 sec. for this problem).





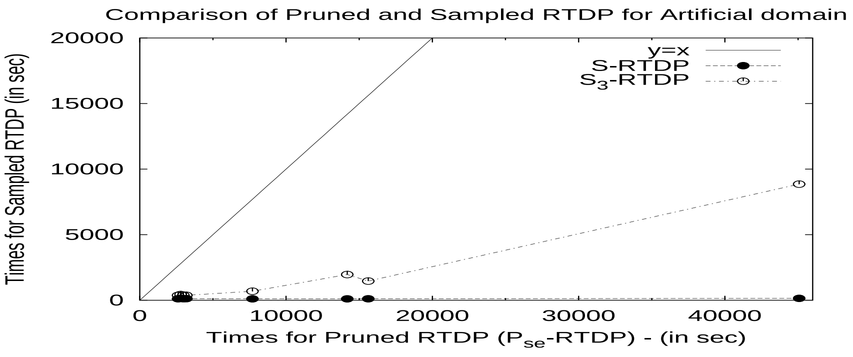
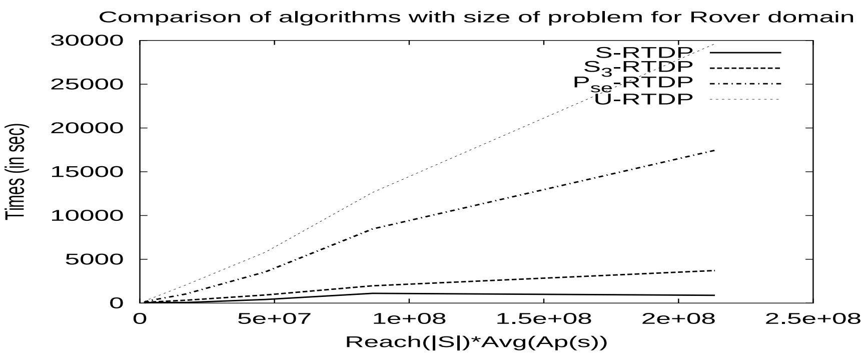
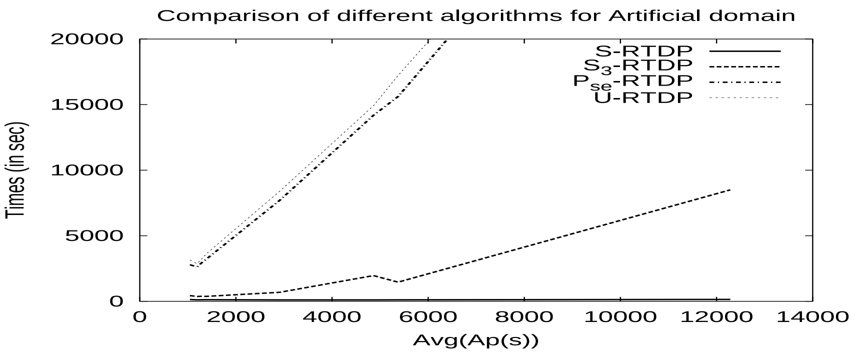


Figure 3: (a,b,c): Sampled vs Pruned RTDP for Rover, Factory and Artificial domains respectively. Random sampling of action combinations yields dramatic improvements in running times.





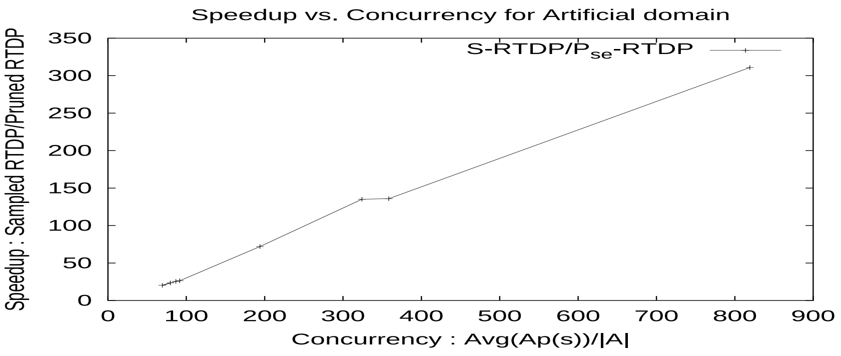


Figure 4: (a,b): Comparison of different algorithms with size of the problems for Rover and Artificial domains. As the problem size increases, the gap between sampled and pruned approaches widens considerably. (c): Relative Speed vs. Concurrency for Artificial domain.

sampled RTDP using both Heuristics 1 and 3, with value functions scaled with 0.9. (S_3-RTDP). We tested all of these algorithms on a number of problem instantiations from our three domains, generated by varying the number of objects, degrees of parallelism, and distances to goal.

Problem	$J(s_0)$ (S-RTDP)	$J^*(s_0)$ (Optimal)	Error
Rover1	10.7538	10.7535	<0.01%
Rover2	10.7535	10.7535	0
Rover3	11.0016	11.0016	0
Rover4	12.7490	12.7461	0.02%
Rover5	7.3163	7.3163	0
Rover6	10.5063	10.5063	0
Rover7	12.9343	12.9246	0.08%
Art1	4.5137	4.5137	0
Art2	6.3847	6.3847	0
Art3	6.5583	6.5583	0
Fact1	15.0859	15.0338	0.35%
Fact2	14.1414	14.0329	0.77%
Fact3	16.3771	16.3412	0.22%
Fact4	15.8588	15.8588	0
Fact5	9.0314	8.9844	0.56%

Table 1: Quality of solutions produced by Sampled RTDP

We observe (Figure 2(a,b)) that pruning significantly speeds the algorithm. But the comparison of P_{se}-RTDP with S-RTDP and S_3-RTDP (Figure 3(a,b,c)) shows that sampling has a dramatic speedup with respect to the pruned versions. In fact, pure sampling, S-RTDP, converges extremely quickly, and S_3-RTDP is slightly slower. However, S_3-RTDP is still much faster than P_{se}-RTDP. The comparison of qualities of solutions produced by S-RTDP and S_3-RTDP *w.r.t.* optimal is shown in Table 1. We observe that solutions produced by S-RTDP are always nearly optimal. Since the error of S-RTDP is small, scaling it by 0.9 makes it an admissible initial value function for the pruned RTDP; indeed, in all experiments, S_3-RTDP produced the optimal solution.

Figure 4(a,b) demonstrates how running times vary with problem size. We use the product of the number of reachable states and the average number of applicable action combinations per state as an estimate of the size of the problem (the number of reachable states in all artificial domains is the same, hence the x-axis for Figure 4(b) is $Avg(Ap(s))$). From these figures, we verify that the number of applicable combinations plays a major role in the running times of the concurrent MDP algorithms. In Figure 4(c), we fix all factors and vary the degree of parallelism. We observe that the speedups obtained by S-RTDP increase as concurrency increases.

This is a very encouraging result, and we can expect S-RTDP to perform well on large problems involving high concurrency, even if the other approaches fail.

In Figure 2(c), we present another experiment in which we vary the number of action combinations sampled in each backup. While solution quality is inferior when sampling only a few combinations, it quickly approaches the optimal on increasing the number of samples. In all other experiments we sample 40 combinations per state.

7 Related Work

Meuleau *et al.* [12] and Singh and Cohn [15] deal with a special type of MDP (called a factorial MDP)[5] that can be represented as a set of smaller weakly coupled MDPs — the separate MDPs are completely independent except for some common resource constraints, and the reward and cost models are purely additive. They describe solutions in which these sub-MDPs are independently solved and the sub-policies are merged to create a global policy. Thus, concurrency of actions of different sub-MDPs is a by-product of their work. Singh & Cohn present an optimal algorithm (similar to our combo-elimination), whereas Meuleau *et al.*'s domain specific heuristics have no such guarantees.

All of the work in Factorial MDPs assumes that a weak coupling exists and has been identified, but factoring an MDP is a hard problem in itself. In contrast, our algorithm can handle strongly coupled MDPs and does not require any sub-task decomposition as input.

Rohanimanesh and Mahadevan [14] investigate a special class of semi-MDPs in which the action space can be partitioned by (possibly concurrent) *Markov options*. They propose an algorithm based on value-iteration, but their focus is calculating joint termination conditions and rewards received, rather than speeding policy construction. Hence, they consider *all* possible Markov option combinations in a backup. Although their model supports options with varying durations, it is restricted in several ways. First, they require the user to specify all possible options (as well as to define the effects of primitive actions). Second, they assume different constraints on action concurrency: i) they omit condition 3 of our mutex definition (Section 3), hence they are subject to race conditions, and ii) their definition restricts some well-defined types of concurrency[6] in a way which may

[5]Guestrin, Koller and Parr [9] have investigated similar representations in the context of multiagent planning.

[6]In their model, two options, o_a and o_b, may not be executed concurrently if there exist actions, $a \in o_a$ and $b \in o_b$, which have 1) inconsistent preconditions or 2) conflicting effects. This is overly conservative because the option's *policies* might guarantee that a and b are never executed concurrently.

preclude finding optimal solutions, which our methods *would* find. Finally, they only experiment on a single, small problem with 400 states.

NASA researchers have developed techniques for solving a harder version of the Rover domain (*e.g.*, with uncertain continuous effects). They propose a *just-in-case* scheduling algorithm, which incrementally adds branches to a straight-line plan. While their work is more general than ours, their solution is heuristic and it is unclear how closely their policies approximate optimality [6, 7]. It would be exciting to combine their methods with ours, perhaps by using their heuristic to guide S-RTDP.

Recently, Younes and Simmons [16] have developed a generic test and debug approach which converts a continuous time MDP into a deterministic planning problem. The optimal plan of the deterministic problem is converted back into a policy which can then be repaired if any failure points are identified.

Fast generation of parallel plans has also been investigated in (deterministic) classical state space based planning scenarios. Edelkamp [8] presents an anytime algorithm that repeatedly creates sequential plans of increasing lengths, and schedules the actions in the plan concurrently using "critical path analysis". This approach is based on the observation that any parallel plan to a goal can be serialised into a valid serial plan to the goal and vice versa. However, this observation is not true in the probabilistic version of the problem as a parallel policy may not be serialisable to a serial policy.

AltAlt^p builds greedy parallelisations within the state space heuristic regression search coupled with *pushing up* the current actions if they can be parallelised with some earlier nodes of the search tree [13]. Unfortunately, its heuristics draw heavily from planning graph constructions that have not been as effective in probabilistic problems. Secondly, as AltAlt^p performs greedy action selection, it is not guaranteed to find an optimal plan.

8 Conclusions and Future Work

This paper formally defines the concurrent MDP problem and describes two algorithms to solve them. *Pruned RTDP* relies on combo-skipping and combo-elimination; with an admissible initial value function, it is guaranteed to converge to an optimal policy and is faster than plain, labeled RTDP on concurrent MDPs. *Sampled RTDP* performs backups on a random subset of possible action combinations; when guided by our heuristics, it converges orders of magnitude faster than other methods and produces optimal or close-to-optimal solutions. We believe that our sampling techniques will be extremely effective on very large, concurrent MDP problems. Moreover, our sampling and pruning techniques are extremely general and can be applied to other base algorithms like value iteration, LAO* *etc.* Thus, we believe our methods will extend easily to solve concurrent MDPs with rewards, non-absorbing goals, and other formulations.

In the future, we wish to prove error bounds on S-RTDP and to modify it so that its convergence is formally guaranteed. Concurrent reinforcement learning may also benefit from our sampling techniques. We also hope to extend our methods to include durative actions, and continuous parameters.

9 Acknowledgements

We are thankful to Blai Bonet and Sumit Sanghai for their inputs at various stages of this research. We also thank David Smith, Subbarao Kambhampati, Stanley Kok, Julie Letchner, Benson Limketkai, Jayant Madhavan, and Parag for giving useful comments on an earlier draft of our AAAI paper [11]. We thank the reviewers for their worthy comments and suggestions on the paper. This work was supported by National Aeronautics and Space Administration (Award NAG 2-1538), National Science Foundation (Award IIS-0307906), and Office of Naval Research (Award N00014-02-1-0932).

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

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

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

[4] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS'03*, pages 12–21. AAAI Press, 2003.

[5] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *J. Artificial Intelligence Research*, 11:1–94, 1999.

[6] John Bresina, Richard Dearden, Nicolas Meuleau, David Smith, and Rich Washington. Planning under continuous time and resource uncertainty : A challenge for AI. In *UAI'02*, pages 77–84, 2002.

[7] Richard Dearden, Nicolas Meuleau, Sailesh Ramakrishnan, David E. Smith, and Rich Washington. Incremental Contingency Planning. In *ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, pages 38–47, 2003.

[8] S. Edelkamp. Taming numbers and duration in the model checking integrated planning system. *Journal of Artificial Intelligence Research*, 20:195–238, 2003.

[9] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *NIPS'01*, pages 1523–1530. The MIT Press, 2001.

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

[11] Mausam and Daniel Weld. Solving concurrent Markov decision processes. In *AAAI'04*, 2004.

[12] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov Decision Processes. In *AAAI'98*, pages 165–172, 1998.

[13] R. S. Nigenda and S. Kambhampati. Altalt-p: Online parallelization of plans with heuristic state search. *Journal of Artificial Intelligence Research*, 19:631–657, 2003.

[14] Khashayar Rohanimanesh and Sridhar Mahadevan. Decision-Theoretic planning with concurrent temporally extended actions. In *UAI'01*, pages 472–479, 2001.

[15] Satinder Singh and David Cohn. How to dynamically merge Markov decision processes. In *NIPS'98*. The MIT Press, 1998.

[16] Håkan L. S. Younes and Reid G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *ICAPS'04*. AAAI Press, 2004.

[17] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI'95*, pages 1114–1120. Morgan Kaufmann, 1995.