Solving Stochastic Shortest-Path Problems with RTDP

Blai Bonet
Cognitive Systems Laboratory
Deptartment of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024

Héctor Geffner
Departamento de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas 1080-A
Venezuela

Abstract

We present a modification of the Real-Time Dynamic Programming (RTDP) algorithm that makes it a genuine off-line algorithm for solving Stochastic Shortest-Path problems. Also, a new domain-independent and admissible heuristic is presented for Stochastic Shortest-Path problems. The new algorithm and heuristic are compared with Value Iteration over benchmark problems with large state spaces. The results show that the modified RTDP algorithm can beat standard Value Iteration by several orders of magnitude in problems with large state space.

Introduction

The class of Stochastic Shortest-Path (SSP) problems is a subset of Markov Decision Processes (MDPs) that is of central importance to AI: they are the *natural* generalization of the classic search model to the case of stochastic transitions and general cost functions. SSPs had been recently used to model a broad range of problems going from robot navigation and control of non-deterministic systems to stochastic game-playing and planning under uncertainty and partial information (Bertsekas & Tsitsiklis 1996; Sutton & Barto 1998; Bonet & Geffner 2000). The theory of MDPs had received great attention from the AI community for three important reasons. First, it provides an easy framework for modeling complex real-life problems that have large state-space (even infinite) and complex dynamics and cost functions. Second, MDPs provide mathematical foundation for independently-developed learning algorithms in Reinforcement Learning. And third, general and efficient algorithms for solving MDPs had been developed, the most important being Value Iteration and Policy Iteration.

As the name suggests, an SSP problem is an MDP problem that has positive costs and an absorbing goal state. A solution for an SSP is a strategy that leads to the goal with minimum expected cost from any other state. Quite often, we are only interested in how to get to the goal from a *fixed* initial state instead of knowing the general solution; the reason being that the state space usually contains many states that are irrelevant. Real-Time Dynamic Programming (RTDP) (Barto, Bradtke, & Singh 1995) is an algorithm for finding such partial solutions. However, RTDP is a *probabilistic* algorithm that only converges *asymptotically*. Hence, although there has been experimental results showing that RTDP converges faster than other algorithms, it cannot be used as a off-line algorithm.

The contribution of this paper is threefold. First, we present a simple method for terminating the RTDP algorithm in finite time while preserving optimality guarantees. The method is formally described, its correctness is proved and a simple implementation is given. Second, we show how to define a general *domain-independent* and admissible heuristic function for SSPs problems that can be used with RTDP (and other search algorithms). The term domain-independent refers to the fact that the definition and computation method are general and work for all SSPs. Finally, we show that the modified RTDP algorithm with the proposed heuristic outperforms the standard Value Iteration algorithm by orders of magnitude in some benchmark problems that have large state spaces.

The paper is organized as follows. The next section contains formal descriptions of the MDP and SSP model, and the RTDP algorithm. In the third section, we show the modification of the RTDP algorithm, prove its correctness and describe a simple implementation. The domain-independent heuristic function is defined in the fourth section along with an explanation of how to *incrementally* compute it within the RTDP algorithm. Then, in the fifth section, the new algorithm and heuristic are tested against Value Iteration. The paper finishes with discussion that includes related work, a summary and future work.

Markov Decision Processes

This section contains a brief review of the MDP, SSP, and RTDP algorithm. We use notation and presentation style as in (Bertsekas 1995); the reader is referred there for a good exposition of the field.

The MDP model assumes the existence of a physical system that evolves in discrete time and that is controlled by an agent. The system dynamics is governed by probabilistic transition functions that maps states and controls to states. At every time, the agent incurs in a cost that depends in the current state of the system and the applied control. Thus, the task is to find a

control strategy (also known as policy) that minimize the expected total cost over the infinite horizon time setting. Formally, an MDP is defined by

(M1) A finite state space $S = \{1, \ldots, n\}$,

(M2) a finite set of controls $U(i)$ for each state $i \in S$,

(M3) transition probabilities $p(i, u, j)$ for all $u \in U(i)$ that are equal to the probability of the next state being j after applying control u in state i, and

(M4) a cost $g(i, u)$ associated to $u \in U(i)$ and $i \in S$.

A strategy or policy π is an infinite sequence $(\mu_0, \mu_1, \ldots)$ of functions where μ_k maps states to controls so that the agent applies the control $\mu_k(i)$ in state $x_k = i$ at time k; the only restriction being that $\mu_k(i) \in U(i)$ for all $i \in S$. If $\pi = (\mu, \mu, \ldots)$, the policy is called *stationary* (i.e., the control does not depend on time) and it is simply denoted by μ. The cost associated to policy π when the system starts at state x_0 is defined as

$$J_\pi(x_0) = \lim_{N \to \infty} E \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(x_k)) \right\} \quad (1)$$

where the expectation is taken with respect to the probability distribution induced by the transition probabilities, and where the number $\alpha \in [0, 1]$, called the *discount factor*, is used to discount future costs at a geometric rate. The MDP *problem* is to find an *optimal policy* π^* satisfying

$$J^*(i) \stackrel{\text{def}}{=} J_{\pi^*}(i) \leq J_\pi(i), \quad i = 1, \ldots, n, \quad (2)$$

for every other policy π. Although there could be none or more than one optimal policy, the optimal cost vector J^* is always unique. The existence of π^* and how to compute it are non-trivial mathematical problems. However, when $\alpha < 1$ the optimal policy always exists and, more important, there exists a stationary policy that is optimal. In such case, J^* is the unique solution to the *Bellman Optimality* equations:

$$J^*(i) = \min_{u \in U(i)} g(i, u) + \alpha \sum_{j=1}^{n} p(i, u, j) J^*(j). \quad (3)$$

Also, if J^* is a solution for (3) then the *greedy* stationary policy μ^* with respect to J^*:

$$\mu^*(i) = \operatorname*{argmin}_{u \in U(i)} \left\{ g(i, u) + \alpha \sum_{j=1}^{n} p(i, u, j) J^*(j) \right\} \quad (4)$$

is an optimal stationary policy for the MDP. Therefore, solving the MDP problem is equivalent to solving (3).

The Value Iteration algorithm computes J^* iteratively by using (3) as an update rule. Indeed, starting from any vector J, Value Iteration computes a succession of vectors $\{J_k\}$ as

$$J_0(i) = J(i) \quad (5)$$

$$J_{k+1}(i) = \min_{u \in U(i)} g(i, u) + \alpha \sum_{j=1}^{n} p(i, u, j) J_k(j). \quad (6)$$

The algorithm stops when $J_{k+1} = J_k$, or when the residual $\max_{i \in S} |J_{k+1}(i) - J_k(i)|$ is sufficiently small. In the latter case, when $\alpha < 1$, the suboptimality of the resulting policy is bounded by a constant multiplied by the residual.

Stochastic Shortest-Path Problems

A Stochastic Shortest-Path problem is an MDP problem in which the state space $S = \{1, \ldots, n, t\}$ is such that t is a goal (target) state that is absorbing (i.e., $p(t, u, t) = 1$ and $g(t, u) = 0$ for all $u \in U(t)$), and the discount factor $\alpha = 1$. In this case, the existence of optimal policies (and optimal stationary policies) is a major mathematical problem. However, the existence is guarantee under the following reasonable conditions:

(A1) There exists a policy that achieves the goal with probability 1 from any starting state.

(A2) All costs are positive.

The first assumption just expresses the fact that the problem admits a well-behaved solution. Such policies are known as *proper* policies. The second assumption, in the other hand, guarantees that all improper policies incurs in infinite cost for at least one state. Thus, both assumptions preclude cases where the optimal solution might "wander" around without never getting to the goal. For example, a problem having a zero-cost cycle (in state space) violates the second assumption.

As mentioned in the Introduction, often we are only interested in knowing how to go from a fixed initial state, say 1, to the goal state. The optimal solution in this case is an *partial* optimal stationary policy μ such that $\mu(i) = \mu^*(i)$ for all states i that are reachable from 1 when using the optimal policy μ^*; the so-called *relevant states* when starting from 1.[1]

Finding a partial optimal policy can be considerably simpler, the extreme case when the set of relevant states is finite and the complete state space is infinite. Thus, the question of how to find partial optimal policies is of great relevance. One algorithm for that is Real-Time Dynamic Programming.

Real-Time Dynamic Programming

The RTDP algorithm is the stochastic generalization of Korf's LRTA* algorithm for heuristic search (Korf 1990). RTDP is a probabilistic algorithm that computes a partial optimal policy by performing successive walks (also called trials) on the state space. Each trial starts at the initial state 1 and finishes at the goal state t. At all times k, the RTDP algorithm maintains an approximation J_k to J^* that is used to *greedly* select a control u_k to apply in the current state x_k. Initially, J_0 is implicitly stored as an heuristic function $h(\cdot)$. Then, every time a control u_k is selected in state x_k, a new

[1] Note that the relevant states are defined with respect to μ^* so any two optimal policies might generate different sets of relevant states. A (stronger) unique definition could involve all optimal policies simultaneously but we don't need that for our purposes.

1. **Initialize** current state $x = 1$,
2. **Evaluate** each control $u \in U(x)$ as
$$Q(x,u) = g(x,u) + \sum_{i \in S} p(x,u,i)J(i)$$
where $J(i) = H(i)$ (resp. $h(i)$) if $i \in H$ (resp. $i \notin H$).
3. **Choose** control $\mathbf{u}$ that minimizes $Q(x,u)$ breaking ties randomly (or in a systematic way),
4. **Update** $H(x)$ to $Q(x,\mathbf{u})$,
5. **Generate** next state i with probability $p(x,\mathbf{u},i)$,
6. **Exit** if $i = t$, else set $x = i$ and go to 2.

Figure 1: A trial of the RTDP algorithm.

approximation J_{k+1} is computed as $J_{k+1}(x) = J_k(x)$ if $x \neq x_k$, and

$$J_{k+1}(x_k) = g(x_k,u_k) + \sum_{i=1}^{n} p(x_k,u_k,i)J_k(i). \quad (7)$$

Since J_k differs from J_0 at most in k states, J_k can be stored efficiently into a hash-table H. Initially, H is empty and the value $H(x)$ is given by $h(x)$, then every time a control is selected an update (7) is applied to H such that J_k can be recovered from H and h. Figure 1 shows a description of an RTDP trial.

It is known that under assumptions A1 and A2, the RTDP trials eventually transverse minimum-cost paths from the initial state to the goal state if the heuristic function is *admissible*; i.e. if $0 \leq h(i) \leq J^*(i)$ for all $i \in S$ (see (Barto, Bradtke, & Singh 1995; Bertsekas & Tsitsiklis 1996)).

The goodness of RTDP had been noted by different researchers in experimental results in which RTDP usually converge faster than Value Iteration in problem withs large state spaces (Barto, Bradtke, & Singh 1995; Hansen & Zilberstein 2001). Unfortunately, the convergence for RTDP holds only *asymptotically* so RTDP is not an authentic off-line algorithm. To make it off-line, we need a method for terminating the trials while preserving some guarantees.

An Off-line RTDP Algorithm

In this section, we present a modification of the RTDP algorithm that terminates the trials when a given precision had been achieved. The modification is called the *stopping rule* and uses a single parameter $\epsilon > 0$. In order to convey the idea, we give two definitions for the method. The first definition is in terms of a global condition that is easy to understand but difficult to implement. The second, in the other hand, is a condition that is more difficult to understand but suggests an easy implementation.

Plainly, the idea is to stop the trials when the value for all relevant states are off from satisfying Bellman equations by at most ϵ; i.e., when

$$\left| J_k(i) - \min_{u \in U(i)} g(i,u) + \sum_{j=1}^{n} p(i,u,j)J_k(j) \right| < \epsilon \quad (8)$$

is satisfied for all relevant states i. The difficulty in testing (8) lies in computing the set of relevant states.

To identify the convergence described by (8), we define a recursive *labeling* that suggests a procedure. The idea is to label states into $\{\text{solved}, \text{unsolved}\}$ such that RTDP terminates when the initial state is labeled as solved. Initially, the goal state is labeled as solved and all other states as unsolved. Let us consider the moment in time just after the end of a trial ($x_0 = 1, \ldots, x_m = t$), and let J be the current approximation function and μ the greedy policy with respect to J. Define the set of states $\{K_j \subseteq S : j = 0, \ldots, m\}$ such that K_j is a minimal set containing x_j and all states reachable from x_j using μ; i.e., K_j is the minimum set of states such that $x_j \in K_j$ and

$$(\forall x \in K_j)(\exists y \in K_j)\big[p(x,\mu(x),y) > 0\big]. \quad (9)$$

Observe that $K_0 \supseteq K_1 \supseteq \cdots \supseteq K_m = \{t\}$. Then, all states in the sets K_j are labeled as solved in the order $j = m-1, \ldots, 0$ until one of the following two conditions fails:

(i) all states in sets K_{j+1} are solved, or

(ii) for all $x \in K_j$

$$\left| J(x) - g(i,\mu(x)) - \sum_{i=1}^{n} p(x,\mu(x),i)J(x) \right| < \epsilon. \quad (10)$$

This labeling is repeated infinitely often at the end of trials until the initial state is labeled as solved. Below we give the proofs that the procedure always finishes for all $\epsilon > 0$ and, more important, that when ϵ is sufficiently small the resulting partial policy is optimal and J is within ϵ distance from J^* for all relevant states.

It is important to note that the RTDP trials do not have to go all the way up to the goal: they can be terminated as soon as a solved state is visited. From now on, when we speak of using the stopping rule we refer to the RTDP algorithm with the stopping rule and trial termination at solved states.

Theorem 1 *For all $\epsilon > 0$ the* RTDP *algorithm with the stopping rule labels the initial state as solved in a finite amount of time.*

Theorem 2 *There exists $\epsilon_0 > 0$ such that if the* RTDP *algorithm with the stopping rule is applied with $0 < \epsilon < \epsilon_0$, then the resulting approximation J satisfies $|J(x) - J^*(x)| < \epsilon$ for all relevant states x, and the resulting partial policy is optimal. Here, $J(x)$ refers to the value for x in the hash-table after termination or $h(x)$ if there is no entry for x in the hash-table.*

Sketches for the proofs are in the Appendix. The labeling procedure can be implemented by different methods: as a time-oriented recursion, as a space-oriented iteration, etc. The following describes the implementation used in the experiments.

Implementation

We decided to implement the stopping rule as an iterative procedure that is applied at the end of a trial each 3 trials. The number 3 is arbitrary and was chosen to reduce the overhead; yet any other frequency will be fine as soon as the procedure is applied infinitely often.

To keep track of the visited states during a trial $T = (x_0, \ldots, x_m)$, RTDP pushes the states into a stack as they are visited. At the end of the trial, the states are processed in reverse visit order as they are popped out from the stack. For each such state x_j, RTDP calls the function `check-solved(`x_j`)` that returns true or false whether conditions (i) and (ii) above are satisfied or not. In the affirmative case, `check-solved(`x_j`)` also returns the set K_j of states that need to be labeled as solved. Otherwise, the stack is cleaned and the process is terminated.

The function `check-solved(`x_j`)` works by applying a breadth-first search that keeps two queues: `open` and `closed`. At the beginning `open` contains only x_j and `closed` is empty. The function then iterates by removing the front state x from `open`, inserting the possible $\mu(x)$ successors of x back into `open` and pushing x into `closed`. Two exceptions are considered:

(a) if the state x is solved, then it is ignored and a new state is removed from `open`, and

(b) if x has never been visited (i.e., it is not in the hash-table) or (10) does not hold for x, then a sequence of hash-table updates (as in Step 4 of Figure 1) is done for state x until (10) is satisfied.

The procedure is applied until `open` becomes empty. Thereafter, `check-solved(`x_j`)` returns true or false whether exception (b) did not occur or did occur respectively. In the former case, the set K_j is the set of states in `closed`. Note that the updates in (b) are only for speeding up the convergence of the algorithm and so they are not necessary for the correctness of the procedure.

Heuristics

The standard technique for getting admissible heuristics for a problem P is to solve a *relaxation* P' of the problem, and then use the solution for P' to compute estimates for P. A good relaxation is one that provides *informative* estimates and is not too difficult to solve. If the estimates are computed carefully, the resulting heuristic is guarantee to be admissible (Pearl 1983).

In our case, we consider the *deterministic* relaxation of the SSP problem defined as the SSP that results by *splitting* each stochastic control u with m outcomes into m deterministic controls. That is, the SSP problem that corresponds to the Bellman equations:

$$\tilde{J}(t) = 0, \tag{11}$$

$$\tilde{J}(i) = \min_{u \in U(i)} g(i,u) + \min\{\tilde{J}(j) : p(i,u,j) > 0\}. \tag{12}$$

The solution $\tilde{J}$ is used to define an admissible heuristic:

$$h(x) = \min_{u \in U(x)} g(x,u) + \sum_{i=1}^{n} p(x,u,i)\tilde{J}(i). \tag{13}$$

The heuristic $h(x)$ can be computed before applying RTDP by the standard Bellman-Ford algorithm (Cormen, Leiserson, & Rivest 1990). However, this method may take long time to finish for problems with large state spaces. Thus, instead of computing the heuristic before applying RTDP, the RTDP trials are *interlaced* with the computation of the heuristic in the following manner: whenever the value $h(x)$ is needed, it is computed by a recursive application of the LRTA* algorithm with the stopping rule over the relaxed problem started at x. The LRTA* procedure uses a separate hash-table and stops when the label for x is solved. Its hash-table is preserved between calls so that if $h(x)$ is needed again, no trials are performed and $h(x)$ is promptly returned. The heuristic for LRTA* is the zero-heuristic. In the next section, we evaluate the new algorithm and heuristic function over some benchmark problems.

Experiments

We tested the algorithm and heuristic vs. the Value Iteration algorithm in a modification of the racetrack domain due to (Barto, Bradtke, & Singh 1995). A problem instance in this domain is defined by a grid-map of a racetrack and a reliability parameter in $[0, 1]$ for the controls. In all the experiments, the reliability was fixed to 0.9. The task consists in driving a car from a set of possible initial states to a set of goal states. Each state in the system is a tuple (x, y, dx, dy) that represents the position and speed of the car in the x, y dimensions. The controls are pairs $u = (ax, ay)$ of instantaneous accelerations where $ax, ay \in \{-1, 0, 1\}$. Thus, the car has momentum and the task is to apply the correct accelerations in order to move the car from the start state to the goal state. When the car hits a wall, the velocities are set to zero and its position is left intact; this is the modification with respect to the original domain in which the car is "magically" translated to the start position after a crash. To give a description of the dynamics, consider the map $proj(s, u)$ from states and controls to states defined by $proj(s, u) = (x + dx + ax, y + dy + ay, dx + ax, dy + ay)$ where $s = (x, y, dx, dy)$ and $u = (ax, ay)$. Then, the transition probabilities are given by

$$p(s,u,s') = \begin{cases} 1 & \text{if } s' = s \wedge s = proj(s,u), \\ 0.9 & \text{if } s \neq s' \wedge s' = proj(s,u), \\ 0.1 & \text{if } s = s' \wedge s' \neq proj(s,u), \\ 0 & \text{otherwise.} \end{cases}$$

We considered the racetracks used in (Barto, Bradtke, & Singh 1995) plus a third one by us. They are shown in Figure 2. In each case, the initial positions correspond to the light-shaded squares while the goal states are the dark-shaded ones.

We made two experiments. In the first, we ran Value Iteration and the RTDP algorithm with the stopping rule using the zero-heuristic (denoted with h_0) and the h heuristic from above. To run Value Iteration, we restricted the possible *infinite* state space to the states of the form (x, y, dx, dy) where $dx \in [-R_x/2, R_x/2]$ and

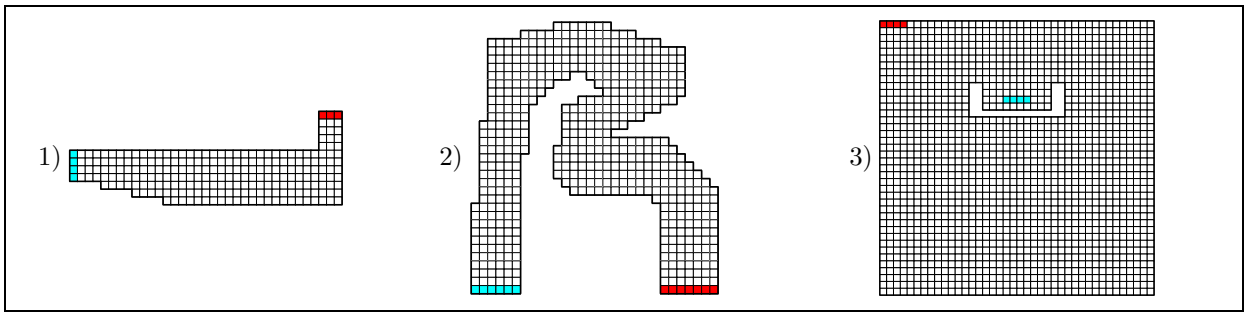

Figure 2: Different racetracks. In each case, the light-shaded squares are the initial states and the dark-shaded ones the goal states.

$dy \in [-R_y/2, R_y/2]$ and R_x (resp. R_y) is the number of columns (resp. rows) in the grid. Table 1 shows the computer time (in seconds) and the number of DP updates needed to achieve a given precision in the problems.[2] The table also shows the size of the state space for Value Iteration (column $|S|$) and the number of visited states for the RTDP algorithm (column $|V|$). As it can be seen, the RTDP algorithms outperform Value Iteration in both total time and number of DP updates by several orders of magnitude. For example, in `race-2`, RTDP is better in time by one order of magnitude and better in DP updates by two orders of magnitude. The situation is far better in `race-3` since Value Iteration can't solve the problem in our machine. In the case of RTDP with the zero-heuristic, we expect that RTDP will visit a considerable portion of the *reachable* state space; i.e., the set of states that can be reached using a finite sequence of controls. Thus, we computed the reachable state space and applied Value Iteration to it. The results are in Table 2. As it can be seen, Value Iteration improves considerably but RTDP is still competitive. For example, Value Iteration cannot beat RTDP in `race-3` even in the reduced state space. In all cases, the number of DP updates is always less for RTDP than for Value Iteration. It is interesting to note how dramatic is the reduction in number of DP updates and relevant states in `race-3` for RTDP with h_0 and h. This result shows that h is a good heuristic for this problem. However, its computation is relatively expensive and that explain the differences in time.

It is important to note that the improvement achieved in Value Iteration by computing the reachable state space will not work in general since quite often the full state space is reachable.

Problem	ϵ	Value Iteration (reachable state space) updates	time	$\|S\|$
RACE-1	10^{-6}	186240	8.4	9312
	10^{-4}	167616	7.2	
	10^{-2}	148992	6.4	
RACE-2	10^{-6}	620906	25.5	23881
	10^{-4}	573144	22.6	
	10^{-2}	525382	20.7	
RACE-3	10^{-6}	3788752	207.0	172216
	10^{-4}	3444320	190.3	
	10^{-2}	2927672	163.1	

Table 2: Results for the second experiment. The Table shows the time (in seconds) and number of DP updates for Value Iteration over the reachable state space.

Discussion

The RTDP algorithm is the stochastic version of Korf's Learning Real-Time A* (LRTA*) algorithm. Clearly, the proposed stopping rule for RTDP also works for the LRTA* algorithm. Others variations of LRTA* had been proposed in the literature; e.g., for speeding up its convergence while preserving some optimality (Ishida & Shimbo 1996), for minimizing worst-case scenarios in multiple outcome decision making (Koenig 2001), for classical planning in dynamic environments (Bonet, Loerincs, & Geffner 1997), etc.

A related algorithm to RTDP for solving SSP problems is the LAO* algorithm of (Hansen & Zilberstein 2001). LAO* also finds partial optimal policies without evaluating the entire space. It is a modification of the standard AO* algorithm for AND/OR graphs (Nilsson 1980) that can cope with cycles and where the AND nodes refer to stochastic transitions. Another recent algorithm for standard AND/OR graphs with cycles is the $\text{CFC}_{\text{REV}^*}$ algorithm of (Jimenéz & Torras 2000). It is an interesting algorithm that also performs a labeling procedure that (we believe) is very close to our procedure. However, its description is more complicate than ours. All other labeling procedures that we know for SSPs assume that the transition graph is acyclic; see (Bertsekas 1995) and references in (Jimenéz & Torras 2000). The basic difference that allows our procedure to cope with cycles is that it can label multiple states *simultaneously* instead of one state at a time. Other incremental algorithms for MDPs are those based on ideas of increasing envelopes (Dean *et al.* 1993).

[2]The experiments were run in a Sun Ultra 10 workstation with 128Mb of memory and a clock rate of 440MHz.

Problem	ϵ	Value Iteration (full state space) updates	time	$\|S\|$	RTDP (heuristic h_0) updates	time	$\|V\|$	RTDP (heuristic h) updates	time	$\|V\|$
RACE-1	10^{-6}	2179480	173.1	108974	95510	8.3	9041	46696	9.4	2140
	10^{-4}	1961532	161.5		84583	7.6	9033	32416	8.1	2135
	10^{-2}	1743584	139.0		71671	5.0	9028	16280	5.7	2148
RACE-2	10^{-6}	16012304	1151.5	571868	888118	181.6	23053	244398	117.5	7006
	10^{-4}	14868568	1021.4		593468	116.4	23097	148429	74.8	7011
	10^{-2}	12581096	865.8		357855	59.3	23068	62407	35.0	7005
RACE-3	10^{-6}	–	–	≥ 2639170	217312	11.4	94513	5130	38.0	372
	10^{-4}	–	–		222367	11.1	96716	3785	37.9	368
	10^{-2}	–	–		210866	10.6	93154	1943	37.5	372

Table 1: Results for the first experiment. The Table shows the time (in seconds) and number of DP updates for the VI and the RTDP algorithm with stopping rule and two different heuristic functions. The columns $|S|$ and $|V|$ refer to the size of the state space in the VI algorithm and the number of visited states in the RTDP algorithms. A dash (–) indicates the algorithm failed due to memory limitations.

As of today, to the best of our knowledge, the LAO* algorithm and RTDP with the stopping rule are the only efficient off-line algorithms for computing optimal partial policies for SSPs. We have deployed the new RTDP algorithm into a general planning tool that is publicly available (** the ref. will be put after review **).

In summary, we presented a new labeling procedure for Stochastic Shortest-Path problems that makes RTDP into an off-line algorithm. We also show a new domain-independent heuristic for SSPs that when used with RTDP outperforms standard Value Iteration in some problems with large state spaces. We believe that, in presence of good heuristic functions, RTDP can be applied to problems with large state spaces (e.g., $> 10^{20}$). Future work will include a through comparison of LAO* and RTDP across different domains and to assess how good is the new heuristic.

References

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

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

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, (2 Vols)*. Athena Scientific.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of AIPS-2000*, 52–61.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*.

Cormen, T.; Leiserson, C.; and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press.

Dean, T.; Kaebling, L.; Kirman, J.; and Nicholson, A. 1993. Planning with deadlines in stochastic domains. In *Proceedings AAAI93*, 574–579.

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

Ishida, T., and Shimbo, M. 1996. Improving the learning efficiencies of realtime search. In *Proceedings of AAAI-96*, 305–310.

Jimenéz, P., and Torras, C. 2000. An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artificial Intelligence* 124:1–30.

Koenig, S. 2001. Minimax real-time heuristic search. *Artificial Intelligence* 129:165–197.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Tioga.

Pearl, J. 1983. *Heuristics*. Morgan Kaufmann.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

Appendix: Proofs

Proof of Theorem 1 (Sketch): For each relevant state x_j consider the set K_j of its reachable states when using the optimal policy μ^*. The collection $\mathcal{K} = \{K_j\}$ is partially ordered by set inclusion. Since the RTDP algorithm converges to the optimal policy and cost vector over the relevant states, then the stopping rule will label as solved all states in a minimal element of $\mathcal{K}$ after some time. After that, the RTDP algorithm with stopping rule will be applying standard RTDP in a reduced MDP consisting of all unsolved states and with terminal costs given by the cost of the solved states. Then, again, the RTDP algorithm converges in this MDP so after some finite time it will label another non-empty set of states as solved. Since the number of states is finite and the initial state is visited infinitely often, the algorithm will label the initial state in finite time. □

Proof of Theorem 2 (Sketch): It is enough to let $\epsilon_0 = \inf\{\|J_\mu - J^*\| : \mu \text{ non-optimal stat. policy}\}$. □