Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming

Blai Bonet
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA
bonet@cs.ucla.edu

Héctor Geffner
Departament de Tecnologia
Universitat Pompeu Fabra
Barcelona 08003, España
hector.geffner@tecn.upf.es

Abstract

RTDP is a recent heuristic-search DP algorithm for solving non-deterministic planning problems with full observability. In relation to other dynamic programming methods, RTDP has two benefits: first, it does not have to evaluate the entire state space in order to deliver an optimal policy, and second, it can often deliver good policies pretty fast. On the other hand, RTDP final convergence is slow. In this paper we introduce a labeling scheme into RTDP that speeds up its convergence while retaining its good anytime behavior. The idea is to label a state s as solved when the heuristic values, and thus, the greedy policy defined by them, have converged over s and the states that can be reached from s with the greedy policy. While due to the presence of cycles, these labels cannot be computed in a recursive, bottom-up fashion in general, we show nonetheless that they can be computed quite fast, and that the overhead is compensated by the recomputations avoided. In addition, when the labeling procedure cannot label a state as solved, it improves the heuristic value of a relevant state. This results in the number of Labeled RTDP trials needed for convergence, unlike the number of RTDP trials, to be bounded. From a practical point of view, Labeled RTDP (LRTDP) converges orders of magnitude faster than RTDP, and faster also than another recent heuristic-search DP algorithm, LAO*. Moreover, LRTDP often converges faster than value iteration, even with the heuristic $h = 0$, thus suggesting that LRTDP has a quite general scope.

Introduction

RTDP is a recent algorithm for solving non-deterministic planning problems with full observability that can be understood both as an heuristic search and a dynamic programming (DP) procedure (Barto, Bradtke, & Singh 1995). RTDP involves simulated greedy searches in which the heuristic values of the states that are visited are updated in a DP fashion, making them consistent with the heuristic values of their possible successor states. Provided that the heuristic is a lower bound (admissible) and the goal is reachable from every state (with positive probability), the updates yield two key properties: first, RTDP trials (runs) are not trapped into loops, and thus must eventually reach the goal, and second, repeated trials yield optimal heuristic values, and hence, an optimal policy, over a closed set of states that includes the initial state (Barto, Bradtke, & Singh 1995; Bertsekas 1995).

RTDP corresponds to a generalization of Korf's LRTA* to non-deterministic settings and is the algorithm that underlies the GPT planner (Bonet & Geffner 2000; 2001a). In relation to other DP algorithms normally used in non-deterministic settings such as Value and Policy Iteration (Bertsekas 1995; Puterman 1994), RTDP has two benefits: first, it is *focused*, namely, it does not have to evaluate the entire state space in order to deliver an optimal policy, and second, it has a good *anytime behavior*; i.e., it produces good policies fast and these policies improve smoothly with time. These features follow from its simulated greedy exploration of the state space: only paths resulting from the greedy policy are considered, and more likely paths, are considered more often. An undesirable effect of this strategy, however, is that unlikely paths tend to be ignored, and hence, RTDP *convergence is slow*.

In this paper we introduce a *labeling scheme* into RTDP that speeds up the convergence of RTDP quite dramatically, while retaining its focus and anytime behavior. The idea is to label a state s as *solved* when the heuristic values, and thus, the greedy policy defined by them, have converged over s and the states that can be reached from s with the greedy policy. While due to the presence of cycles, these labels cannot be computed in a recursive, bottom-up fashion in general, we show nonetheless that they can be computed quite fast, and that the overhead is compensated by the recomputations avoided. In addition, when the labeling procedure fails to label a state as solved, it always improves the heuristic value of a relevant state. As a result, the number of Labeled RTDP trials needed for convergence, unlike the number of RTDP trials, is bounded. From a practical point of view, we show empirically over some problems that Labeled RTDP (LRTDP) converges orders of magnitude faster than RTDP, and faster also than another recent heuristic-search DP algorithm, LAO* (Hansen & Zilberstein 2001). Moreover, in our experiments, LRTDP converges faster than value iteration even with the heuristic function $h = 0$, thus suggesting LRTDP may have a quite general scope.

The paper is organized as follows. We cover first the basic model for non-deterministic problems with full feedback, and then value iteration and RTDP (Section 2). We then assess the anytime and convergence behavior of these two methods (Section 3), introduce the labeling procedure into RTDP, and discuss its theoretical properties (Section 4). We finally run an empirical evaluation of the resulting algorithm (Section 5) and end with a brief discussion (Section 6).

Preliminaries

We provide a brief review of the models needed for characterizing non-deterministic planning tasks with full observability, and some of the algorithms that have been proposed for solving them. We follow the presentation in (Bonet & Geffner 2001b); see (Bertsekas 1995) for an excellent exposition of the field.

Deterministic Models with No Feedback

Deterministic state models (Newell & Simon 1972; Nilsson 1980) are the most basic action models in AI and consist of a finite set of states S, a finite set of actions A, and a state transition function f that describes how actions map one state into another. Classical planning problems can be understood in terms of deterministic state models comprising:

S1. a discrete and finite state space S,

S2. an initial state $s_0 \in S$,

S3. a non empty set $G \subseteq S$ of goal states,

S4. actions $A(s) \subseteq A$ applicable in each state $s \in S$,

S5. a deterministic state transition function $f(s, a)$ for $s \in S$ and $a \in A(s)$,

S6. positive action costs $c(a, s) > 0$.

A *solution* to a model of this type is a sequence of actions $a_0, a_1, \ldots, a_n$ that generates a state trajectory s_0, $s_1 = f(s_0, a_0), \ldots, s_{n+1} = f(s_i, a_i)$ such that each action a_i is applicable in s_i and s_{n+1} is a goal state, i.e. $a_i \in A(s_i)$ and $s_{n+1} \in G$. The solution is *optimal* when the total cost $\sum_{i=0}^{n} c(s_i, a_i)$ is minimal.

Non-Deterministic Models with Feedback

Non-deterministic planning problems with full feedback can be modeled by replacing the deterministic transition function $f(s, a)$ in S5 by *transition probabilities* $P_a(s'|s)$,[1] resulting in the model:[2]

M1. a discrete and finite state space S,

M2. an initial state $s_0 \in S$,

M3. a non empty set $G \subseteq S$ of goal states,

M4. actions $A(s) \subseteq A$ applicable in each state $s \in S$,

M5. transition probabilities $P_a(s'|s)$ of reaching state s' after aplying action $a \in A(s)$ in $s \in S$,

M6. positive action costs $c(a, s) > 0$.

[1]Namely, the numbers $P_a(s'|s)$ are real and non-negative and their sum over the states $s' \in S$ must be 1.

[2]The ideas in the paper apply also, with slight modification, to non-deterministic planning where transitions are modeled by means of functions $F(s, a)$ mapping s and a into a non-empty *set* of states (e.g., (Bonet & Geffner 2000)).

While the states s_{i+1} that result from an action a_i are *not predictable* they are assumed to be *fully observable*, providing *feedback* for the selection of the action a_{i+1}:

M7. states are fully observable.

The models M1–M7 are known as Markov Decision Processes or MDPs, and more specifically as *Stochastic Shortest-Path Problems* (Bertsekas 1995).[3]

Due to the presence of feedback, the solution of an MDP is not an action sequence but a function π mapping states s into actions $a \in A(s)$. Such a function is called a *policy*. A policy π assigns a probability to every state trajectory $s_0, s_1, s_2, \ldots$ starting in state s_0 which is given by the product of all transition probabilities $P_{a_i}(s_{i+1}|s_i)$ with $a_i = \pi(s_i)$. If we further assume that actions in goal states have no costs and produce no changes (i.e., $c(a, s) = 0$ and $P_a(s|s) = 1$ if $s \in G$), the *expected cost* associated with a policy π starting in state s_0 is given by the weighted average of the probability of such trajectories multiplied by their cost $\sum_{i=0}^{\infty} c(\pi(s_i), s_i)$. An *optimal solution* is a control policy π^* that has a minimum expected cost for all possible initial states. An optimal solution for models of the form M1–M7 is guaranteed to exist when the goal is reachable from every state $s \in S$ with some positive probability (Bertsekas 1995). This optimal solution, however, is not necessarily unique.

We are interested in computing optimal and near-optimal policies. Moreover, since we assume that the initial state s_0 of the system is known, it is sufficient to compute a *partial policy that is closed* relative to s_0. A partial policy π only prescribes the action $\pi(s)$ to be taken over a subset $S_\pi \subseteq S$ of states. If we refer to the states that are reachable (with some probability) from s_0 using π as $S(\pi, s_0)$ (including s_0), then a partial policy π is closed relative to s_0 when $S(\pi, s_0) \subseteq S_\pi$. We sometimes refer to the states reachable by an *optimal policy* as the *relevant states*. Traditional dynamic programming methods like value or policy iteration compute complete policies, while recent heuristic search DP methods like RTDP and LAO* are more *focused* and aim to compute only closed, partial optimal policies over an smaller set of states that includes the relevant states. Like more traditional heuristic search algorithms, they achieve this by means of suitable heuristic functions $h(s)$ that provide non-overestimating (admissible) estimates of the expected cost to reach the goal from any state s.

Dynamic Programming

Any heuristic function h defines a *greedy policy* π_h:

$$\pi_h(s) \;=\; \operatorname*{argmin}_{s \in A(s)} \; c(a, s) + \sum_{s' \in S} P_a(s'|s)\; h(s') \qquad (1)$$

where the expected cost from the resulting states s' is assumed to be given by $h(s')$. We call $\pi_h(s)$ the *greedy action* in s for the heuristic or value function h. If we denote the optimal (expected) cost from a state s to the goal by $V^*(s)$,

[3]For discounted and other MDP formulations, see (Puterman 1994; Bertsekas 1995). For uses of MDPs in AI, see for example (Sutton & Barto 1998; Boutilier, Dean, & Hanks 1995; Barto, Bradtke, & Singh 1995; Russell & Norvig 1994).

it is well known that the greedy policy π_h is *optimal* when h is the *optimal cost function*, i.e. $h = V^*$.

While due to the possible presence of ties in (1), the greedy policy is not unique, we assume throughout the paper that these ties are broken systematically using an static ordering on actions. As a result, every value function V defines a *unique* greedy policy π_V, and the *optimal* cost function V^* defines a unique optimal policy π_{V^*}. We define the *relevant states* as the states that are reachable from s_0 using this optimal policy; they constitute a minimal set of states over which the optimal value function needs to be defined.[4]

Value iteration is a standard dynamic programming method for solving non-deterministic states models such as M1–M7 and is based on computing the optimal cost function V^* and plugging it into the greedy policy (1). This optimal cost function is the unique solution to the fixed point equation (Bellman 1957):

$$V(s) \; = \; \min_{a \in A(s)} c(a,s) + \sum_{s' \in S} P_a(s'|s)\, V(s') \qquad (2)$$

also called the Bellman equation. For stochastic shortest path problems like M1-M7 above, the border condition $V(s) = 0$ is also needed for goal states $s \in G$. Value iteration solves (2) for V^* by plugging an initial guess in the right-hand of (2) and obtaining a new guess on the left-hand side. In the form of value iteration known as *asynchronous value iteration* (Bertsekas 1995), this operation can be expressed as:

$$V(s) \; := \; \min_{a \in A(s)} c(a,s) + \sum_{s' \in S} P_a(s'|s)\, V(s') \qquad (3)$$

where V is a vector of size $|S|$ initialized arbitrarily (except for $s \in G$ initialized to $V(s) = 0$), and where we have replaced the equality in (2) by an assignment. The use of expression (3) for updating a state value in V is called a state update or simply an *update*. In standard (synchronous) value iteration, all states are updated in parallel, while in asynchronous value iteration, only a selected subset of states is selected for update. In both cases, it is known that if all states are updated infinitely often, the value function V converges eventually to the optimal value function. In the case that concerns us, model M1–M7, the only assumption that is needed for convergence is that (Bertsekas 1995)

A. *the goal is reachable with positive probability from every state*.

From a practical point of view, value iteration is stopped when the Bellman error or *residual* defined as the difference between left and right sides of (2):

$$R(s) \stackrel{\text{def}}{=} \left| V(s) - \min_{a \in A(s)} c(a,s) + \sum_{s' \in S} P_a(s'|s)\, V(s') \right| \quad (4)$$

over all states s is sufficiently small. In the discounted MDP formulation, a bound on the policy loss (the difference between the expected cost of the policy and the expected cost of an optimal policy) can be obtained as a simple expression of the discount factor and the maximum residual. In stochastic shortest path models such as M1–M7, there is no similar closed-form bound, although such bound can be computed at some expense (Bertsekas 1995; Hansen & Zilberstein 2001). Thus, one can execute value iteration until the maximum residual becomes smaller than a bound ϵ, then if the bound on the policy loss is higher than desired, the same process can be repeated with a smaller ϵ (e.g., $\epsilon/2$) and so on. For these reasons, we will take as our *basic task* below, the computation of a value function $V(s)$ with residuals (4) smaller than or equal to some given parameter $\epsilon > 0$ for all states. Moreover, since we assume that the initial state s_0 is known, it will be enough to check the residuals over all states that can be reached from s_0 using the greedy policy π_V.

[4]This definition of 'relevant states' is more restricted than the one in (Barto, Bradtke, & Singh 1995) that includes the states reachable from s_0 by *any* optimal policy.

```
RTDP(s : state)
begin
    repeat RTDPTrial(s)
end

RTDPTrial(s : state)
begin
    while ¬s.GOAL() do
        // pick best action and update hash
        a = s.GREEDYACTION()
        s.UPDATE()
        // stochastically simulate next state
        s = s.PICKNEXTSTATE(a)
end
```

Algorithm 1: RTDP algorithm (with no termination conditions; see text)

One last definition and a few known results before we proceed. We say that cost vector V is *monotonic* iff

$$V(s) \; \leq \; \min_{a \in A(s)} c(a,s) + \sum_{s' \in S} P_a(s'|s)\, V(s')$$

for every $s \in S$. The following are well-known results (Barto, Bradtke, & Singh 1995; Bertsekas 1995):

Theorem 1 *a) Given the assumption A, the optimal values $V^*(s)$ of model M1-M7 are non-negative and finite; b) the admissibility and monotonicity of V are preserved through updates.*

Real-Time Dynamic Programming

RTDP is a simple DP algorithm that involves a sequence of trials or runs, each starting in the initial state s_0 and ending in a goal state (Algorithm 1, some common routines are defined in Alg.2). Each RTDP trial is the result of simulating the greedy policy π_V while updating the values $V(s)$ using (3) over the states s that are visited. Thus, RTDP is an asynchronous value iteration algorithm in which a single state is selected for update in each iteration. This state corresponds to the state visited by the simulation where successor states are selected with their corresponding probability. This combination is powerful, and makes RTDP different

$state$ :: GREEDYACTION()
begin
  return $\underset{a \in A(s)}{\operatorname{argmin}}\ s.\text{QVALUE}(a)$
end

$state$ :: QVALUE(a : $action$)
begin
  return $c(s,a) + \sum_{s'} P_a(s'|s) \cdot s'.\text{VALUE}$
end

$state$ :: UPDATE()
begin
  $a = s.\text{GREEDYACTION}()$
  $s.\text{VALUE} = s.\text{QVALUE}(a)$
end

$state$ :: PICKNEXTSTATE(a : $action$)
begin
  pick s' with probability $P_a(s'|s)$
  return s'
end

$state$ :: RESIDUAL()
begin
  $a = s.\text{GREEDYACTION}()$
  return $|s.\text{VALUE} - s.\text{QVALUE}(a)|$
end

Algorithm 2: Some state routines.

than pure greedy search and general asynchronous value iteration. Proofs for some properties of RTDP can be found in (Barto, Bradtke, & Singh 1995), (Bertsekas 1995) and (Bertsekas & Tsitsiklis 1996); a model like M1–M7 is assumed.

Theorem 2 (Bertsekas and Tsitsiklis (1996)) *If assumption A holds,* RTDP *unlike the greedy policy cannot be trapped into loops forever and must eventually reach the goal in every trial. That is, every* RTDP *trial terminates in a finite number of steps.*

Theorem 3 (Barto et al. (1995)) *If assumption A holds, and the initial value function is* admissible *(lower bound), repeated* RTDP *trials eventually yield optimal values $V(s) = V^*(s)$ over all relevant states.*

Note that unlike asynchronous value iteration, in RTDP there is *no* requirement that all states be updated infinitely often. Indeed, some states may not be updated at all. On the other hand, the proof of optimality lies in the fact that the *relevant* states will be so updated.

These results are very important. Classical heuristic search algorithms like IDA* can solve optimally problems with trillion or more states (e.g. (Korf & Taylor 1996)), as the use of lower bounds (admissible heuristic functions) allows them to bypass most of the states in the space. RTDP is the first DP method that offers the same potential in a general non-deterministic setting. Of course, in both cases, the quality of the heuristic functions is crucial for performance, and indeed, work in classical heuristic search has often been aimed at the discovery of new and effective heuristic functions (see (Korf 1997)). LAO* (Hansen & Zilberstein 2001) is a more recent heuristic-search DP algorithm that shares this feature.

For an efficient implementation of RTDP, a hash table T is needed for storing the updated values of the value function V. Initially the values of this function are given by an heuristic function h and the table is empty. Then, every time that the value $V(s)$ for a state s that is not in the table is updated, and entry for s in the table is created. For states s in the table T, $V(s) = T(s)$, while for states s that are not in the table, $V(s) = h(s)$.

The convergence of RTDP is asymptotic, and indeed, the number of trials before convergence is not bounded (e.g., low probability transitions are taken eventually but there is always a positive probability they will not be taken after any arbitrarily large number of steps). Thus for practical reasons, like for value iteration, we define the *termination* of RTDP in terms of a given bound $\epsilon > 0$ on the residuals (no termination criterion for RTDP is given in (Barto, Bradtke, & Singh 1995)). More precisely, we terminate RTDP when the value function converges over the initial state s_0 relative to a positive ϵ parameter where

Definition 1 *A value function V* converges *over a state s relative to parameter $\epsilon > 0$ when the residuals* (4) *over s and all states s' reachable from s by the greedy policy π_V are smaller than or equal to ϵ.*

Provided with this termination condition, it can be shown that as ϵ approaches 0, RTDP delivers an optimal cost function and an optimal policy over all relevant states.

Convergence and Anytime Behavior

From a practical point of view, RTDP has positive and negative features. The positive feature is that RTDP has a good *anytime behavior*: it can quickly produce a good policy and then smoothly improve it with time. The negative feature is that *convergence*, and hence, termination is slow. We illustrate these two features by comparing the anytime and convergence behavior of RTDP in relation to value iteration (VI) over a couple of experiments. We display the anytime behavior of these methods by plotting the average cost to the goal of the greedy policy π_V as a function of time. This cost changes in time as a result of the updates on the value function. The averages are taken by running 500 simulations after every 25 trials for RTDP, and 500 simulations after each syncrhonous update for VI. The curves in Fig. 1 are further smoothed by a moving average over 50 points (RTDP) and 5 points (VI).

The domain that we use for the experiments is the racetrack from (Barto, Bradtke, & Singh 1995). A problem in this domain is characterized by a racetrack divided into cells (see Fig. 2). The task is to find the control for driving a car from a set of initial states to a set of goal states minimizing

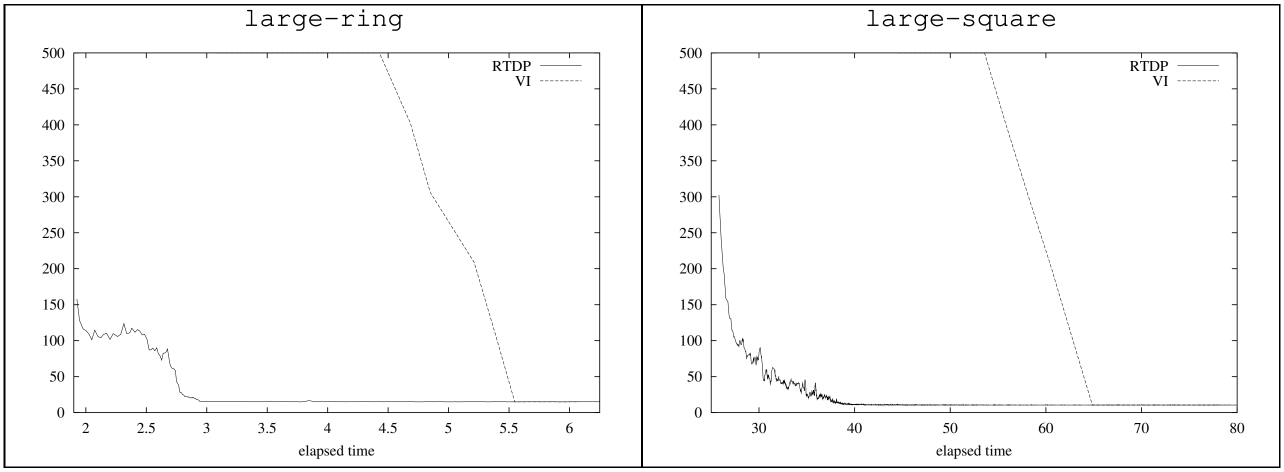

Figure 1: Quality profiles: Average cost to the goal vs. time for Value Iteration and RTDP over two racetracks.

the number of time steps.[5]

The states are tuples (x, y, dx, dy) that represent the position and speed of the car in the x, y dimensions. The actions are pairs $a = (ax, ay)$ of instantaneous accelerations where $ax, ay \in \{-1, 0, 1\}$. Uncertainty in this domain comes from assuming that the road is 'slippery' and as a result, the car may fail to accelerate. More precisely, an action $a = (ax, ay)$ has its intended effect 90% of the time; 10% of the time the action effects correspond to those of the action $a_0 = (0, 0)$. Also, when the car hits a wall, its velocity is set to zero and its position is left intact (this is different than in (Barto, Bradtke, & Singh 1995) where the car is moved to the start position).

For the experiments we consider racetracks of two shapes: one is a ring with 33068 states (`large-ring`), the other is a full square with 383950 states (`large-square`). In both cases, the task is to drive the car from one extreme in the racetrack to its opposite extreme. Optimal policies for these problems turn out to have expected costs (number of steps) 14.96 and 10.48 respectively, and the percentages of relevant states (i.e., those reachable through an optimal policy) are 11.47% and 0.25% respectively.

Figure 1 shows the quality of the policies found as a function of time for both value iteration and RTDP. In both cases we use $V = h = 0$ as the initial value function; i.e., no (informative) heuristic is used in RTDP. Still, the quality profile for RTDP is much better: it produces a policy that is practically as good as the one produced eventually by VI in almost half of the time. Moreover, by the time RTDP yields a policy with an average cost close to optimal, VI yields a policy with an average cost that is more than 10 times higher.

The problem with RTDP, on the other hand, is *convergence*. Indeed, while in these examples RTDP produces near-to-optimal policies at times 3 and 40 seconds respectively, it converges, in the sense defined above, in more than 10 *minutes*. Value iteration, on the other hand, converges in 5.435 and 78.720 seconds respectively.

[5]So far we have only considered a single initial state s_0, yet it is simple to modify the algorithm and extend the theoretical results for the case of multiple initial states.

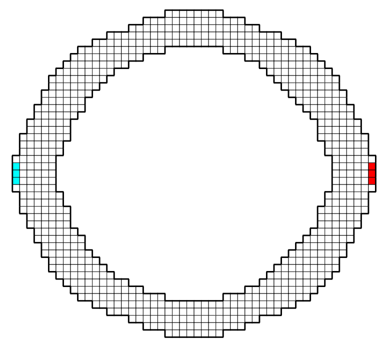

Figure 2: Racetrack for `large-ring`. The initial and goal positions marked on the left and right

Labeled RTDP

The fast improvement phase in RTDP as well as its slow convergence phase are both consequences of an exploration strategy – greedy simulation – that privileges the states that occur in the most likely paths resulting from the greedy policy. These are the states that are most relevant given the current value function, and that's why updating them produces such a large impact. Yet states along less likely paths are also needed for convergence, but these states appear less often in the simulations. This suggests that a potential way for improving the convergence of RTDP is by adding some *noise* in the *simulation*.[6] Here, however, we take a different approach that has interesting consequences both from a practical and a theoretical point of view.

[6]This is different than adding noise in the action selection rule as done in reinforcement learning algorithms (Sutton & Barto 1998) In reinforcement learning, noise is needed in the action selection rule to guarantee optimality while no such thing is needed in RTDP. This is because full DP updates as used in RTDP, unlike partial DP updates as used in Q or TD-learning, preserve admissibility.

Roughly, rather than forcing RTDP out of the most likely greedy states, we keep track of the states over which the value function has converged and avoid visiting those states again. This is accomplished by means of a *labeling procedure* that we call CHECKSOLVED. Let us refer to the graph made up of the states that are reachable from a state s with the greedy policy (for the current value function V) as the *greedy graph* and let us refer to the set of states in such a graph as the *greedy envelope*. Then, when invoked in a state s, the procedure CHECKSOLVED searches the greedy graph rooted at s for a state s' with residual greater than ϵ. If no such state s' is found, and only then, the state s is labeled as SOLVED and CHECKSOLVED(s) returns **true**. Namely, CHECKSOLVED(s) returns **true** and labels s as SOLVED when the current value function has converged over s. In that case, CHECKSOLVED(s) also labels as SOLVED all states s' in the greedy envelope of s as, by definition, they must have converged too. On the other hand, if a state s' is found in the greedy envelope of s with residual greater than ϵ, then this and possibly other states are updated and CHECKSOLVED(s) returns **false**. These updates are crucial for accounting for some of the theoretical and practical differences between RTDP and the labeled version of RTDP that we call *Labeled* RTDP or LRTDP.

Note that due to the presence of cycles in the greedy graph it is not possible in general to have the type of recursive, bottom-up labeling procedures that are common in some AO* implementations (Nilsson 1980), in which a state is labeled as solved when its successors states are solved.[7] Such a labeling procedure would be sound but incomplete, namely, in the presence of cycles it may fail to label as solved states over which the value function has converged.

The code for CHECKSOLVED is shown in Alg. 3. The label of state s in the procedure is represented with a bit in the hash table denoted with s.SOLVED. Thus, a state s is labeled as solved iff s.SOLVED = **true**. Note that the Depth First Search stores all states visited in a *closed* list to avoid loops and duplicate work. Thus the time and space complexity of CHECKSOLVED(s) is $O(|S|)$ in the worst case, while a tighter bound is given by $O(|S_V(s)|)$ where $S_V(s) \subseteq S$ refers to the greedy envelope of s for the value function V. This distinction is relevant as the greedy envelope may be much smaller than the complete state space in certain cases depending on the domain and the quality of the initial value (heuristic) function.

An additional detail of the CHECKSOLVED procedure shown in Alg. 3, is that for performance reasons, the search in the greedy graph is not stopped when a state s' with residual greater than ϵ is found; rather, all such states are found and collected for update in the *closed* list. Still such states act as fringe states and the search does not proceed beneath them (i.e., their children in the greedy graph are not added to the *open* list).

```
CHECKSOLVED(s : state, ε : float)
begin
    rv = true
    open = EMPTYSTACK
    closed = EMPTYSTACK
    if ¬s.SOLVED then open.PUSH(s)

    while open ≠ EMPTYSTACK do
        s = open.POP()
        closed.PUSH(s)

        // check residual
        if s.RESIDUAL() > ε then
            rv = false
            continue

        // expand state
        a = s.GREEDYACTION()
        foreach s' such that P_a(s', s) > 0 do
            if ¬s'.SOLVED & ¬IN(s', open ∪ closed)
                open.PUSH(s')

    if rv = true then
        // label relevant states
        foreach s' ∈ closed do
            s'.SOLVED = true
    else
        // update states with residuals and ancestors
        while closed ≠ EMPTYSTACK do
            s = closed.POP()
            s.UPDATE()
    return rv
end
```

Algorithm 3: CHECKSOLVED

In presence of a monotonic value function V, DP updates have always a *non-decreasing* effect on V. As result, we can establish the following key property:

Theorem 4 *Assume that V is an admissible and monotonic value function. Then, a* CHECKSOLVED(s, ϵ) *call either labels a state as solved, or increases the value of some state by more than ϵ while decreasing the value of none.*

A consequence of this is that CHECKSOLVED can solve by itself models of the form M1-M7. If we say that a model is *solved* when a value function V converges over the initial state s_0, then it is simple to prove that:

Theorem 5 *Provided that the initial value function is admissible and monotonic, and that the goal is reachable from every state, then the procedure:*

while $\neg s_0$.SOLVED **do** CHECKSOLVED(s_0, ϵ)

solves the model M1-M7 in a number of iterations no greater than $\epsilon^{-1} \sum_{s \in S} V^*(s) - h(s)$, where each iteration has complexity $O(|S|)$.

[7] The use of labeling procedures for solving AND-OR graphs with cyclic solutions is hinted in (Hansen & Zilberstein 2001), yet the authors do not mention that the same reasons that preclude the use of backward induction procedures for computing state values in AND-OR graphs with cyclic solutions, also preclude the use of bottom up, recursive labeling procedures as found in standard AO* implementations.

```
LRTDP(s : state, ε : float)
begin
    while ¬s.SOLVED do LRTDPTRIAL(s, ε)
end

LRTDPTRIAL(s : state, ε : float)
begin
    visited = EMPTYSTACK
    while ¬s.SOLVED do
        // insert into visited
        visited.PUSH(s)

        // check termination at goal states
        if s.GOAL() then break

        // pick best action and update hash
        a = s.GREEDYACTION()
        s.UPDATE()

        // stochastically simulate next state
        s = s.PICKNEXTSTATE(a)

    // try labeling visited states in reverse order
    while visited ≠ EMPTYSTACK do
        s = visited.POP()
        if ¬CHECKSOLVED(s, ε) then break
end
```

Algorithm 4: LRTDP.

The solving procedure in Theorem 5 is novel and interesting although it is not sufficiently practical. The problem is that it does never attempt to label other states as *solved* before labeling s_0. Yet, unless the greedy envelope is a strongly connected graph, s_0 will be the last state to converge and not the first. In particular, states that are close to the goal will normally converge faster than states that are farther.

This is the motivation for the alternative use of the CHECKSOLVED procedure in the *Labeled* RTDP *algorithm*. LRTDP trials are very much like RTDP trials except that trials terminate when a *solved* stated is reached (initially only the goal states are solved) and they invoke then the CHECKSOLVED procedure in reverse order, from the last *unsolved* state visited in the trial back to s_0, until the procedure returns **false** on some state. Labeled RTDP terminates when the initial state s_0 is labeled as SOLVED. The code for LRTDP is shown in Alg. 4. The first property of LRTDP is inherited from RTDP:

Theorem 6 *Provided that the goal is reachable from every state and the initial value function is admissible, then* LRTDP *trials cannot be trapped into loops, and hence must terminate in a finite number of steps.*

The novelty in the optimality property of LRTDP is the bound on the total number of trials required for convergence

Theorem 7 *Provided that the goal is reachable from every state and the initial value function is admissible and monotonic, then* LRTDP *solves the model M1-M7 in a number of trials bounded by* $\epsilon^{-1}\sum_{s\in S} V^*(s) - h(s)$.

Empirical Evaluation

In this section we evaluate LRTDP empirically in relation to RTDP and two other DP algorithms: value iteration, the standard dynamic programming algorithm, and LAO*, a recent extension of the AO* algorithm introduced in (Hansen & Zilberstein 2001) for dealing with AND-OR graphs with cyclic solutions. Value iteration is the baseline for performance; it is a very practical algorithm yet it computes complete policies and needs as much memory as states in the state space. LAO*, on the other hand, is an heuristic search algorithm like RTDP and LRTDP that computes partial optimal policies and does not need to consider the entire space. We are interested in comparing these algorithms along two dimensions: anytime and convergence behavior. We evaluate anytime behavior by plotting the average cost of the policies found by the different methods as a function of time, as described in the section on convergence and anytime behavior. Likewise, we evaluate convergence by displaying the time and memory (number of states) required.

We perform this analysis with a domain-independent heuristic function for initializing the value function and with the heuristic function $h = 0$. The former is obtained by solving a simple relaxation of the Bellman equation

$$V(s) \;=\; \min_{a\in A(s)} c(a,s) + \sum_{s'\in S} P_a(s'|s)\,V(s') \qquad (5)$$

where the expected value taken over the possible successor states is replaced by the min such value:

$$V(s) \;=\; \min_{a\in A(s)} c(a,s) + \min_{s':P_a(s'|s)>0} V(s') \qquad (6)$$

Clearly, the optimal value function of the relaxation is a lower bound, and we compute it on need by running a labeled variant of the LRTA* algorithm (Korf 1990) (the deterministic variant of RTDP) until convergence, from the state s where the heuristic value $h(s)$ is needed. Once again, unlike standard DP methods, LRTA* does not require to evaluate the entire state space in order to compute these values, and moreover, a separate hash table containing the values produced by these calls is kept in memory, as these values are reused among calls (this is like doing several single-source shortest path problems on need, as opposed to a single but potentially more costly all-sources shortest-path problem when the set of states is very large). The total time taken for computing these heuristic values in the heuristic search procedures like LRTDP and LAO*, as well as the quality of these estimates, will be shown separately below. Briefly, we will see that this time is significant and often more significant than the time taken by LRTDP. Yet, we will also see that the overall time, remains competitive with respect to value iteration and to the same LRTDP algorithm with the $h = 0$ heuristic. We call the heuristic defined by the relaxation above, h_{min}. It is easy to show that this heuristic is admissible and monotonic.

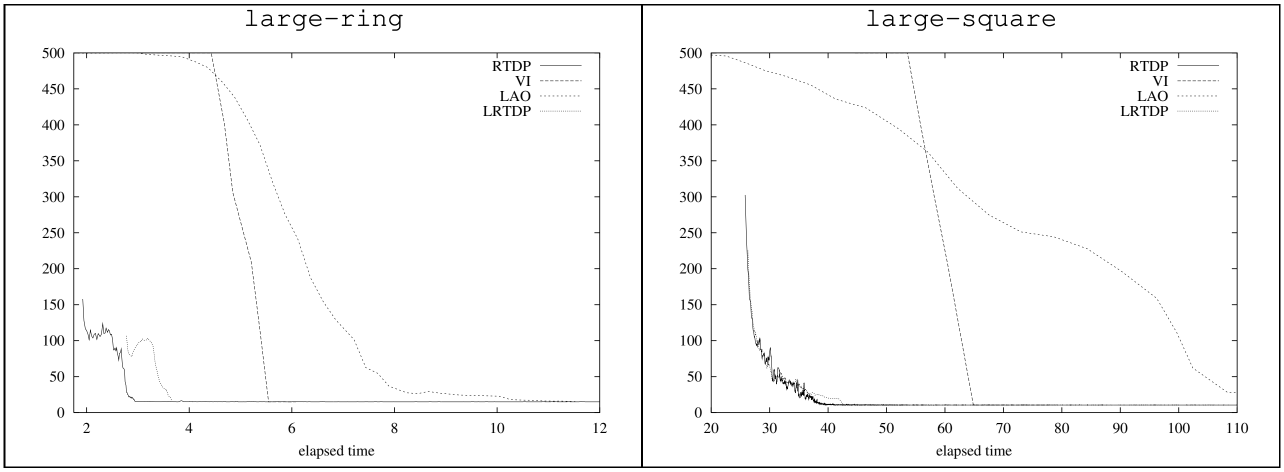

Figure 3: Quality profiles: Average cost to the goal vs. time for RTDP, VI, ILAO* and LRTDP with the heuristic $h = 0$ and $\epsilon = 10^{-3}$.

problem	$\|S\|$	$V(s_0)$	% rel.	$h_{min}(s_0)$	t. h_{min}
small-b	9312	11.084	13.96	10.00	0.740
large-b	23880	17.147	19.36	16.00	2.226
h-track	53597	38.433	17.22	36.00	8.714
small-r	5895	10.377	10.65	10.00	0.413
large-r	33068	14.960	11.47	14.00	3.035
small-s	42071	7.508	1.67	7.00	3.139
large-s	383950	10.484	0.25	10.00	41.626
small-y	81775	13.764	1.35	13.00	8.623
large-y	239089	15.462	0.86	14.00	29.614

Table 1: Information about the different ractrack instances: size, expected cost of the optimal policy from s_0, percentage of relevant states, heuristic for s_0 and time spent in seconds for computing h_{min}.

Problems

The problems we consider are all instances of the racetrack domain from (Barto, Bradtke, & Singh 1995) discussed earlier. We consider the instances small-b and large-b from (Barto, Bradtke, & Singh 1995), h-track from (Hansen & Zilberstein 2001),[8] and three other tracks, ring, square, and y (small and large) with the corresponding shapes. Table 1 contains relevant information about these instances: their size, the expected cost of the optimal policy from s_0, and the percentage of relevant states.[9] The last two columns show information about the heuristic h_{min}: the value for $h_{min}(s_0)$ and the total time involved in the computation of heuristic values in the LRTDP and LAO* algorithms for the different instances. Interestingly, these times are roughly equal for both algorithms across the different problem instances. Note that the heuristic provides a pretty tight lower bound yet it is somewhat expensive. Below, we will run the experiments with both the heuristic h_{min} and the heuristic $h = 0$.

[8] Taken from the source code of LAO*.

[9] All these problems have multiple initial states so the value $V^*(s_0)$ (and $h_{min}(s_0)$) in the table is the average over the intial states.

Algorithms

We consider the following four algorithms in the comparison: VI, RTDP, LRTDP, and a variant of LAO*, called Improved LAO* in (Hansen & Zilberstein 2001). Due to cycles, the standard backward induction step for backing up values in AO* is replaced in LAO* by a full DP step. Like AO*, LAO* gradually grows an explicit graph or envelope, originally including the state s_0 only, and after every iteration, it computes the best policy over this graph. LAO* stops when the graph is closed with respect to the policy. From a practical point of view, LAO* is slow; Improved LAO* (ILAO*) is a variation of LAO* in (Hansen & Zilberstein 2001) that gives up on some of the properties of LAO* but runs much faster.[10]

The four algorithms have been implemented by us in C++. The results have been obtained on a Sun-Fire 280-R with 1GB of RAM and a clock speed of 750Mhz.

Results

Curves in Fig. 3 display the evolution of the average cost to the goal as a function of time for the different algorithms, with averages computed as explained before, for two problems: large-ring and large-square. The curves correspond to $\epsilon = 10^{-3}$ and $h = 0$. RTDP shows the best profile, quickly producing policies with average costs near optimal, with LRTDP close behind, and VI and ILAO* farther.

Table 2 shows the times needed for convergence (in seconds) for $h = 0$ and $\epsilon = 10^{-3}$ (results for other values of ϵ are similar). The times for RTDP are not reported as they exceed the cutoff time for convergence (10 minutes) in all instances. Note that for the heuristic $h = 0$, LRTDP converges faster than VI in 6 of the 8 problems, with problem h-track, LRTDP running behind by less than 1% of the

[10] ILAO* is presented in (Hansen & Zilberstein 2001) as an implementation of LAO*, yet it is a different algorithm with different properties. In particular, ILAO*, unlike, LAO* does not maintain an optimal policy over the explicit graph across iterations.

algorithm	small-b	large-b	h-track	small-r	large-r	small-s	large-s	small-y	large-y
VI($h = 0$)	1.101	**4.045**	**15.451**	0.662	5.435	5.896	78.720	16.418	61.773
ILAO*($h = 0$)	2.568	11.794	43.591	1.114	11.166	12.212	250.739	57.488	182.649
LRTDP($h = 0$)	**0.885**	7.116	15.591	**0.431**	**4.275**	**3.238**	**49.312**	**9.393**	**34.100**

Table 2: Convergence time in seconds for the different algorithms with initial value function $h = 0$ and $\epsilon = 10^{-3}$. Times for RTDP not shown as they exceed the cutoff time for convergence (10 minutes). Faster times are shown in bold font.

algorithm	small-b	large-b	h-track	small-r	large-r	small-s	large-s	small-y	large-y
VI(h_{min})	1.317	4.093	12.693	0.737	5.932	6.855	102.946	17.636	66.253
ILAO*(h_{min})	1.161	2.910	11.401	0.309	3.514	0.387	1.055	0.692	1.367
LRTDP(h_{min})	**0.521**	**2.660**	**7.944**	**0.187**	**1.599**	**0.259**	**0.653**	**0.336**	**0.749**

Table 3: Convergence time in seconds for the different algorithms with initial value function $h = h_{min}$ and $\epsilon = 10^{-3}$. Times for RTDP not shown as they exceed the cutoff time for convergence (10 minutes). Faster times are shown in bold font.

time taken by VI. ILAO* takes longer yet also solves all problems in a reasonable time. The reason that LRTDP, like RTDP, can behave so well even in the absence of an informative heuristic function is because the focused search and updates quickly boost the values that appear most relevant so that they soon provide such heuristic function, an heuristic function that has not been given but has been learned in the sense of (Korf 1990). The difference between the heuristic-search approaches such as LRTDP and ILAO* on the one hand, and classical DP methods like VI on the other, is more prominent when an informative heuristic like h_{min} is used to seed the value function. Table 3 shows the corresponding results, where the first three rows show the convergence times for VI, ILAO* and LRTDP with the initial values obtained with the heuristic h_{min} but with the time spent computing such heuristic values excluded. An average of such times is displayed in the last column of Table 1, as these times are roughly equal for the different methods (the standard deviation in each case is less than 1% of the average time). Clearly, ILAO* and LRTDP make use of the heuristic information much more effectively than VI, and while the computation of the heuristic values is expensive, LRTDP with h_{min}, taking into account the time to compute the heuristic values, remains competitive with VI with $h = 0$ and with LRTDP with $h = 0$. E.g., for `large-square` the overall time for the former is $0.653 + 41.626 = 42.279$ while the latter two times are 78.720 and 49.312. Something similar occurs in `small-y` and `small-ring`. Of course, these results could be improved by speeding up the computation of the heuristic h_{min}, something we are currently working on.

Summary

We have introduced a labeling scheme into RTDP that speeds up its convergence while retaining its good anytime behavior. While due to the presence of cycles, the labels cannot be computed in a recursive, bottom-up fashion as in standard AO* implementations, they can be computed quite fast by means of a search procedure that when cannot label a state as solved, improves the value of some states by a finite amount. The labeling procedure has interesting theoretical and practical properties. On the theoretical side, the number of Labeled RTDP trials, unlike the number of RTDP trials is bounded; on the practical side, Labeled RTDP converges much faster than RTDP, and appears to converge faster than Value Iteration and LAO*, while exhibiting a better anytime profile. Also, Labeled RTDP often converges faster than VI even with the heuristic $h = 0$, suggesting that the proposed algorithm may have a quite general scope.

We are currently working on alternative methods for computing or approximating the heuristic h_{min} proposed, and variations of the labeling procedure for obtaining practical methods for solving larger problems.

Acknowledgments: We thank Eric Hansen and Shlomo Zilberstein for making the code for LAO* available to us. Blai Bonet is supported by grants from NSF, ONR, AFOSR, DoD MURI program, and by a USB/CONICIT fellowship from Venezuela.

References

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

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

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 Chien, S.; Kambhampati, S.; and Knoblock, C., eds., *Proc. 5th International Conf. on Artificial Intelligence Planning and Scheduling*, 52–61. Breckenridge, CO: AAAI Press.

Bonet, B., and Geffner, H. 2001a. GPT: a tool for planning with uncertainty and partial information. In Cimatti, A.; Geffner, H.; Giunchiglia, E.; and Rintanen, J., eds., *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, 82–87.

Bonet, B., and Geffner, H. 2001b. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence* 14(3):237–252.

Boutilier, C.; Dean, T.; and Hanks, S. 1995. Planning under uncertainty: structural assumptions and computational leverage. In *Proceedings of EWSP-95*.

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

Korf, R., and Taylor, L. 1996. Finding optimal solutions to the twenty-four puzzle. In Clancey, W., and Weld, D., eds., *Proc. 13th National Conf. on Artificial Intelligence*, 1202–1207. Portland, OR: AAAI Press / MIT Press.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.

Korf, R. 1997. Finding optimal solutions to rubik's cube using patterns databases. In Kuipers, B., and Webber, B., eds., *Proc. 14th National Conf. on Artificial Intelligence*, 700–705. Providence, RI: AAAI Press / MIT Press.

Newell, A., and Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice–Hall.

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

Puterman, M. 1994. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc.

Russell, S., and Norvig, P. 1994. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

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