Learning Depth-First Search: A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and its application to MDPs

Blai Bonet
Departamento de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Héctor Geffner
Departamento de Tecnología
ICREA & Universitat Pompeu Fabra
Barcelona 08003, Spain
hector.geffner@upf.edu

Abstract

Dynamic Programming provides a convenient and unified framework for studying many state models used in AI but no algorithms for handling large spaces. Heuristic-search methods, on the other hand, can handle large spaces but lack a common foundation. In this work, we combine the benefits of a general dynamic programming formulation with the power of heuristic-search techniques for developing an algorithmic framework, that we call Learning Depth-First Search, that aims to be both general and effective. LDFS is a simple piece of code that performs iterated depth-first searches enhanced with learning. For deterministic actions and monotone value functions, LDFS reduces to IDA* with transposition tables, while for Game Trees, to the state-of-the-art iterated Alpha-Beta search algorithm with Null Windows known as MTD. For other models, like AND/OR graphs and MDPs, LDFS yields new, simple, and competitive algorithms. We show this here for MDPs.

Introduction

Dynamic Programming provides a convenient and unified framework for studying many state models used in AI (Bellman 1957; Bertsekas 1995) but no algorithms for handling large spaces. Heuristic-search methods, on the other hand, can handle large spaces effectively, but lack a common foundation: algorithms like IDA* aim at deterministic models (Korf 1985), AO* at AND/OR graphs (Martelli & Montanari 1973), Alpha-Beta at Game Trees (Newell, Shaw, & Simon 1963), and so on (Nilsson 1980; Pearl 1983). In this work, we aim to combine the benefits of a general dynamic programming formulation with the effectiveness of heuristic-search techniques for developing an algorithmic framework, that we call *Learning Depth-First Search* (LDFS) that aims to be both general and effective. For some models, LDFS reduces to well-known algorithms, like IDA* with transposition tables for Deterministic Models (Reinefeld & Marsland 1994) and MTD for Game Trees (Plaat *et al.* 1996), while for other models, like AND/OR Graphs and Markov Decision Processes, it yields new and competitive algorithms.

The LDFS framework is built around two simple notions: depth-first search and *learning* in the sense of (Korf 1990) and (Barto, Bradtke, & Singh 1995), where state values are updated to make them consistent with the values of their successors. LDFS shows that the key idea underlying several existing heuristic-search algorithms over various settings is *the use of iterated depth-first searches with memory for improving the heuristics over certain relevant states until they become optimal.* Also, by generalizing these algorithms, LDFS carries this idea to other settings where it results in a novel and effective approach.

LDFS grows from recent work on MDPs that combines DP updates with the use of lower bounds and knowledge of the initial state for computing *partial* optimal policies over the relevant states efficiently (Barto, Bradtke, & Singh 1995; Hansen & Zilberstein 2001; Bonet & Geffner 2003b; 2003a; McMahan, Likhachev, & Gordon 2005). However, rather than focusing on the development of another heuristic-search MDP algorithm, we make use of these notions to lay out a general framework covering a wide range of models which we intend to be general, transparent, and useful. This generality pays off in a number of ways; for example, by showing that a given MDP algorithm reduces to IDA* when all probabilities are 0 or 1, it becomes clear that the MDP algorithm is doing certain things right and is not missing key features. This is important as heuristic-search algorithms for MDPs are not as well established and mature as heuristic-search algorithms for deterministic problems. Similarly, we will be able to explain the weaknesses of some MDP algorithms in terms of well-known weaknesses of IDA*; for example, the potential blow up in the number of iterations when action costs are real numbers rather than integers (Wah & Shang 1994).

While Hansen and Zilberstein offer a generalization of AO* to MDPs based on the general notion of *best-first search* (Hansen & Zilberstein 2001), LDFS provides a generalization of IDA* based on the notions of *depth-first search* and *learning*, that covers deterministic problems, AND/OR graphs (cyclic or not), MDPs, and that at the same time extends to Game Trees where it provides a clear and explicit correspondence between IDA* and the state-of-the-art MTD algorithm (Plaat *et al.* 1996).

The paper is organized as follows. We deal first with a common formulation of the various models and a simple common algorithmic schema, Find-and-Revise, that solves them all. We then introduce LDFS as an efficient instance of this schema for models that have acyclic solutions. We then move to MDPs, where the LDFS algorithm is extended and evaluated.

Models

All the state models that we consider can be defined in terms of the following elements:

1. a discrete and finite state space S,
2. an initial state $s_0 \in S$,
3. a non-empty set of terminal states $S_T \subseteq S$,
4. actions $A(s) \subseteq A$ applicable in each non-terminal state,
5. a function mapping non-terminal states s and actions $a \in A(s)$ into *sets* of states $F(a, s) \subseteq S$,
6. action costs $c(a, s)$ for non-terminal states s, and
7. terminal costs $c_T(s)$ for terminal states.

We assume that both $A(s)$ and $F(a, s)$ are non-empty. The various models correspond to:

- Deterministic Models (DET): $|F(a, s)| = 1$,
- Non-Deterministic Models (NON-DET): $|F(a, s)| \geq 1$,
- Markov Decision Processes (MDPs): with probabilities $P_a(s'|s)$ for $s' \in F(a, s)$ s.t. $\sum_{s' \in F(a,s)} P_a(s'|s) = 1$.

In addition, for DET, NON-DET, and MDPs

- action costs $c(a, s)$ are all positive, and
- terminal costs $c_T(s)$ are non-negative.

When terminal costs are all zero, terminal states are called *goals*. AND/OR graphs can be modeled as non-deterministic state models, while Game Trees (GT) correspond to non-deterministic state models with zero action costs and arbitrary terminal costs.

For distinguishing models with and without cycles, let a path $s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n$ be a sequence of states and actions starting in the initial state s_0, such that each action a_i is applicable in s_i, $a_i \in A(s_i)$, and each state s_{i+1} is a possible successor of s_i given action a_i, $s_{i+1} \in F(a_i, s_i)$. Then a model is *cyclic* if it can give rise to *cyclic paths*; namely, paths $s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n$ with $s_i = s_j$ for $i \neq j$. Otherwise, a model is *acyclic*. We write aNON-DET and aMDPs to refer to the subclass of acyclic NON-DET and MDPs models. For example, the type of problems in the scope of the AO* algorithm corresponds to those in aNON-DET. It is useful to note that only MDPs can have cyclic solutions; all the other models, whether cyclic or not, can only have solutions without cycles. More about this below.

Solutions

The solutions to the various models can be expressed in terms of the so-called Bellman equation that characterizes the optimal cost function (Bellman 1957; Bertsekas 1995):

$$V(s) \overset{\text{def}}{=} \begin{cases} c_T(s) & \text{if } s \text{ terminal} \\ \min_{a \in A(s)} Q_V(a, s) & \text{otherwise} \end{cases} \qquad (1)$$

where the $Q_V(a, s)$ values express the cost-to-go and are *short-hand* for:

$c(a, s) + V(s')$, $s' \in F(a, s)$ for DET,
$c(a, s) + \max_{s' \in F(a,s)} V(s')$ for NON-DET-Max,
$c(a, s) + \sum_{s' \in F(a,s)} V(s')$ for NON-DET-Add,
$c(a, s) + \sum_{s' \in F(a,s)} P_a(s'|s)V(s')$ for MDPs, and
$\max_{s' \in F(a,s)} V(s')$ for Game Trees.

We will refer to the models (NON-DET-Max and GT) whose Q-values are defined with Max as Max models, and to the rest of the models, defined with Sums, as Additive models.

Under some conditions, there is a unique value function $V^*(s)$, the optimal cost function, that solves the Bellman equation, and the optimal solutions to all the models can be expressed in terms of the policies π that are *greedy* with respect to $V^*(s)$. A policy π is a function mapping states $s \in S$ into actions $a \in A(s)$, and a policy π_V is greedy with respect to a value function $V(s)$, or simply greedy in V, iff π_V is the best policy assuming that the cost-to-go is given by $V(s)$; i.e.

$$\pi_V(s) \; = \; \underset{a \in A(s)}{\operatorname{argmin}}\, Q_V(a, s) \, . \qquad (2)$$

Often, however, these conditions are not met, and the set of $|S|$ Bellman equations have no solution. This happens for example in the presence of *dead-ends*, whether or not such dead-end states can be avoided on the way to the goal. The absence of dead-ends is a common requirement in DP methods and algorithms like LRTA* (Korf 1990) and RTDP (Barto, Bradtke, & Singh 1995) that demand the goal to be reachable (with some positive probability) from every state. Here, we drop this requirement by focusing on *partial policies* that map a subcollection of states into actions. We say that a partial policy π is *closed* (relative to s_0) if π prescribes the action to be done in all the (non-terminal) *states reachable by π from s_0*. In particular, closed policies for deterministic models correspond to action sequences, for game trees, to actual trees, and so on.

Any closed policy π relative to a state s has a cost $V^\pi(s)$ that expresses the cost of solving the problem following π starting from s. The costs $V^\pi(s)$ are given by the solution of (1) but with the operator $\min_{a \in A(s)}$ removed and the action a replaced by $\pi(s)$. These costs are well-defined when the resulting equations have a solution *over the subset of states reachable by π from s_0*. Moreover, for all models above, except MDPs, it can be shown that (closed) policies π have a well-defined finite cost $V^\pi(s_0)$ when they are *acyclic*, and for MDPs, when they are *proper*. Otherwise $V^\pi(s_0) = \infty$. A (closed) policy π is *cyclic* if it gives rise to cyclic paths $s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n$ where $a_i = \pi(s_i)$, and it is *proper* if a terminal state is reachable with some probability from every state s reachable by π from s_0 (Bertsekas 1995).

For all models except for MDPs, since solutions π are acyclic, the costs $V^\pi(s_0)$ can be defined also recursively, starting with the terminal states s' for which $V^\pi(s') = c_T(s')$, and up to the non-terminal states s for which $V^\pi(s) = Q_{V^\pi}(\pi(s), s)$. In all cases, we are interested in computing a solution π that minimizes $V^\pi(s_0)$. The resulting value is the optimal problem cost $V^*(s_0)$.

Computing Solutions

We assume throughout the paper that we have an initial value function (or heuristic) V that is a *monotone lower bound*, i.e., $V(s) \leq V^*(s)$ and $V(s) \leq \min_{a \in A(s)} Q_V(a,s)$. Also for simplicity, we assume $V(s) = c_T(s)$ for all *terminal* states. We summarize these conditions by simply saying that V is **admissible.** This value function is then modified by *learning* in the sense of (Korf 1990) and (Barto, Bradtke, & Singh 1995), where the values of selected states are made consistent with the values of successor states; an operation that takes the form of a Bellman *update*:

$$V(s) := \min_{a \in A(s)} Q_V(a,s) . \qquad (3)$$

If the initial value function is admissible, it remains so after one or more updates. Methods like *value iteration* perform the iteration $V(s) := \min_{a \in A(s)} Q_V(a,s)$ over all states until the difference between right and left does not exceed some $\epsilon \geq 0$. The difference $\min_{a \in A(s)} Q_V(a,s) - V(s)$, which is non-negative for monotone value functions, is called the *residual* of V over s, denoted $Res_V(s)$. Clearly, a value function V is a solution to Bellman equation and is thus equal to V^* if it has zero residuals over all states. Given a fixed initial state s_0, however, it is not actually necessary to eliminate all residuals for ensuring optimality:

Proposition 1 *Let V be an admissible value function and let π be a policy greedy in V. Then π minimizes $V^\pi(s_0)$ and hence is optimal if $Res_V(s) = 0$ over all the states s reachable from s_0 and π.*

This suggests a simple and general schema for solving all the models, that avoids the strong assumptions required by standard DP methods and yields partial optimal policies. It is actually sufficient to search for a state s reachable from s_0 and π_V with residual $Res_V(s) > \epsilon$ and update the state, keeping this loop until there are no such states left. If $\epsilon = 0$ and the initial value function V is admissible, then the resulting (closed) greedy policy π_V is optimal. This FIND-and-REVISE schema, shown in Fig. 1 and introduced in (Bonet & Geffner 2003a) for solving MDPs, can be used to solve all the models without having to compute complete policies:[1]

Proposition 2 *Starting with an admissible value function V, the FIND-and-REVISE schema for $\epsilon = 0$, solves all the models (DET, NON-DET, GT, MDPs) provided they have solution.*

For the non-probabilistic models with *integer* action and terminal costs, the number of iterations of FIND-and-REVISE with $\epsilon = 0$ can actually be bounded by $\sum_{s \in S}[V^*(s) - V(s)]$ when there are no dead-ends (states with $V^*(s) = \infty$), as the updates increase the value function by a positive integer in some states and decrease it at none, preserving its admissibility. In the presence of dead-ends, the bound can be set to $\sum_{s \in S}[\min(V^*(s), MaxV) - V(s)]$ where $MaxV$ stands for any upper bound on the optimal costs $V^*(s)$ of the states with *finite cost*, as states with values above $MaxV$ will not be reachable by an optimal policy from s_0. Since the Find procedure can be implemented by a simple DFS procedure that keeps track of visited states in time $O(|S|)$, it follows that the time complexity of FIND-and-REVISE over those models can be bounded by this expression times $O(|S|)$. For MDPs, the convergence of FIND-and-REVISE with $\epsilon = 0$ is asymptotic and cannot be bounded in this way. However, for any $\epsilon > 0$, the convergence is bounded by the same expression divided by ϵ.

[1]We assume that the initial value function V (which may be the zero function) is represented intensionally and that the updated values are stored in a hash table. Also, since there may be many policies π greedy in V, we use π_V to refer to the unique greedy policy obtained by selecting in each state the action greedy in V that is minimal according to some fixed ordering on actions.

```
starting with an admissible $V$
repeat
    FIND $s$ reachable from $s_0$ and $\pi_V$ with $Res_V(s) > \epsilon$
    Update $V(s)$ to $\min_{a \in A(s)} Q_V(a,s)$
until no such state is found
return $V$
```

Figure 1: The FIND-and-REVISE schema

LDFS

All the models considered above admit a common formulation and a common algorithm, FIND-and-REVISE. This algorithm, while not practical, is useful for understanding and proving the correctness of other, more effective approaches. We will say that an iterative algorithm is instance of FIND-and-REVISE$[\epsilon]$, if each iteration of the algorithm terminates, either identifying and updating a state reachable from s_0 and π_V with residual $Res_V(s) > \epsilon$, or proving that no such state exists, and hence, that the model is solved. Such algorithms will inherit the correctness of FIND-and-REVISE, but by performing more updates per iteration will converge faster.

We focus first on the models whose *solutions* are necessarily acyclic, excluding thus MDPs but not acyclic MDPs (aMDPs). We are not excluding *models* with cycles though; only models whose *solutions* may be cyclic. Hence the requirements are weaker than those of algorithms like AO*.

We will say that a state s is *consistent* relative to a value function V if the residual of V over s is no greater than ϵ. Unless mentioned otherwise, we take ϵ to be 0. The first practical instance of FIND-and-REVISE that we consider, LDFS, implements the Find operation as a DFS that starts in s_0 and recursively descends into the successor states $s' \in F(a,s)$ iff a is an applicable action in s and the value function V is such that $Q_V(a,s) \leq V(s)$. Since V is assumed to be monotone, and therefore $Q_V(a,s) \geq V(s)$, one such action a exists if and only if s is consistent. Thus, if there is no such action a at s, s is inconsistent, and the LDFS algorithm backtracks on s, updating s and its ancestors.

The code for LDFS is shown in Fig. 2. The Depth-First Search is achieved by means of two loops: one over the actions $a \in A(s)$, and the other, nested, over the possible successors $s' \in F(a,s)$. The recursion occurs when a is such that $Q_V(a,s) \not> V(s)$, which given that V is assumed to be a monotone function, is exactly when s is consistent and a is a greedy action in s. The tip nodes in the search are the inconsistent states s (where for all the actions

```
LDFS-DRIVER($s_0$)
begin
    repeat $solved := \text{LDFS}(s_0)$ until $solved$
    return $(V, \pi)$
end

LDFS($s$)
begin
    if $s$ is SOLVED or terminal then
        if $s$ is terminal then $V(s) := c_T(s)$
        Mark $s$ as solved return $true$

    $flag := false$
    foreach $a \in A(s)$ do
        if $Q_V(a,s) > V(s)$ then continue
        $flag := true$
        foreach $s' \in F(a,s)$ do
            $flag := \text{LDFS}(s')\ \&\ flag$          % Recursion
        if $flag$ then break

    if $flag$ then
        $\pi(s) := a$
        Mark $s$ as SOLVED                          % Labeling
    else
        $V(s) := \min_{a \in A(s)} Q_V(a,s)$        % Update
    return $flag$
end
```

Figure 2: Learning DFS Algorithm (LDFS)

$Q_V(a,s) > V(s)$), the terminal states, and the states that are labeled as solved. A state s is labeled as solved when the search beneath s does not find an inconsistent state. This is captured by the boolean $flag$. If s is consistent, and $flag$ is true after searching beneath the successors $s' \in F(a,s)$ of a greedy action a at state s, then s is labeled as solved, $\pi(s)$ is set to a, and no more actions are tried at s. Otherwise, the next action is tried, and if no one is left, s is updated, and $false$ is returned. Solved states are not explored again and become tip nodes in the search.

LDFS is called iteratively over s_0 from a driver routine that terminates when s_0 is solved, returning a value function V and a greedy policy π that satisfies Proposition 1, and hence is optimal. We show this by proving that LDFS is an instance of FIND-and-REVISE. First, since no model other than MDPs can accommodate *a cycle of consistent states*, we get that:

Proposition 3 *For DET, NON-DET, GT, and aMDPs, a call to* LDFS *cannot enter into a loop and thus terminates.*

Then, provided with the same ordering on actions as FIND-and-REVISE, it is simple to show that the first state s that is updated by LDFS is inconsistent and reachable from s_0 and π_V, and if there is not such state, LDFS terminates with the policy π_V.

Proposition 4 *Provided an initial admissible value function,* LDFS *is an instance of* FIND-*and*-REVISE$[\epsilon = 0]$, *and hence, it terminates with a closed partial policy π that is optimal for DET, NON-DET, GT, and aMDPs.*

In addition, for the models that are *additive*, it can be shown that all the updates performed by LDFS are *effective*, in the sense that they are all done on states that are inconsistent, and which as a result, strictly increase their values. More precisely:

Proposition 5 *Every recursive call* LDFS(s) *over the* additive models *DET, NON-DET-Add, and aMDPs either increases the value of s or labels s as solved.*

An immediate consequence of this is that for DET and NON-DET-Add models with integer action and terminal costs, the number of iterations can be bounded by $V^*(s_0) - V(s_0)$ as all value increases must be integer too. This bound is tighter than the one for FIND-and-REVISE and corresponds exactly to the maximum number of iterations in IDA* under the same conditions. Actually, provided that LDFS and IDA* (with transposition tables, Reinefeld & Marsland 1994) consider the actions in the same order, it can be shown that they will both traverse the same paths, and maintain the same value (heuristic) function in memory:

Proposition 6 (LDFS & IDA*) *Provided an admissible (and monotone) value function V, and that actions are applied in the same order in every state,* LDFS *is equivalent to* IDA* *with Transposition Tables (Reinefeld & Marsland 1994) over the class of deterministic models (DET).*

This result may seem puzzling at first because the code for LDFS, while more general than IDA* with transposition tables, is also simpler. In particular, unlike IDA*, LDFS does not need to carry an explicit bound as argument. This simplification however follows from the assumption that the value function V is monotone. In such a case, and provided that transposition tables and cost revision are used as in (Reinefeld & Marsland 1994), the condition $f(s) = g(s) + V(s) > Bound$ for pruning a node in IDA*, where $g(s)$ is the accumulated cost and $V(s)$ is the estimated cost to the goal, becomes simply that s is inconsistent.

In order to get an intuition for this, notice that LDFS traverses a path $s_0, a_0, s_1, a_1, \ldots, s_k, a_k$ only when s_{i+1} is a successor of a_i in s_i and $Q_V(a_i, s_i) \leq V(s_i)$, where $Q_V(a_i, s_i)$ for DET is $c(a_i, s_i) + V(s_{i+1})$. Now, for a monotone V, $c(a_i, s_i) + V(s_{i+1}) \leq V(s_i)$ holds iff the equality $c(a_i, s_i) + V(s_{i+1}) = V(s_i)$ holds, which if applied iteratively, yields that the path is traversed only if $V(s_0) = g(s_i) + V(s_i)$ for $i = 0, \ldots, k$, where $g(s_i)$ is the accumulated cost from s_0 to s_i along the path. For a monotone V, this condition is equivalent to $V(s_0) \geq g(s_i) + V(s_i)$, which is the exact complement of the pruning condition $g(s_i) + V(s_i) > Bound$ in IDA* when $Bound = V(s_0)$, something that is true in IDA* when V is monotone and transposition tables are used.

Actually, if we consider a binary tree where the nodes are states s that admit two deterministic actions mapping s into its two sons respectively, we can get a simple characterization of the workings of both LDFS and IDA* with transposition tables. Let us assume $V(s) = 0$ for all states, and that the goals are the leaves at depth n. Then in the first trial, LDFS finds that s_0 is an inconsistent state (according to V) and thus prunes the search beneath s_0 and updates $V(s_0)$ to

1. This update makes s_0 consistent, and hence, in the second trial, the tip nodes of the search become the states at depth 1 whose values are inconsistent, and which are made consistent by updating them to 1. At the same time, upon backtracking, the consistency of s_0 is restored, updating its value to 2. The algorithms keeps working in this way, and by the time the iteration i is finished, with $0 \leq i < n$, the value function V is consistent over all the states at depth $j \leq i$. Namely, at that time, nodes at depth i have value 1, nodes at depth $i-1$ have value 2, and so on, while the single root node s_0 at depth 0 has value $V(s_0) = i$. The same result holds for IDA* with transposition tables.

LDFS generalizes the key idea underlying IDA* to other models, which enter into the LDFS algorithm through the $Q_V(a, s)$ term whose form depends on the model. Indeed, for all Additive Models, the behavior of LDFS can be characterized as follows:

Proposition 7 *Over the Additive Models DET, Non-DET-Add, and aMDPs,* LDFS *tests whether there is a solution* π *with cost* $V^{\pi}(s_0) \leq V(s_0)$ *for an initial admissible value function* V*. If a solution exists, one such solution is found and reported; else* $V(s_0)$ *is increased, and the test is run again, til a solution is found. Since* V *remains a lower bound, the solution found is optimal.*

This is the key idea underlying not only IDA* but also the Memory-enhanced Test Driver algorithm or MTD$(-\infty)$ for Game Trees (Plaat *et al.* 1996). Both algorithms work as 'Test and Revise' loops; namely, lower bounds are used as upper bounds, and if no solution is found, the lower bounds are increased accordingly, and the loop is resumed.

Interestingly, while LDFS solves Game Trees and Max AND/OR Graphs, it does not exhibit this 'Test and Revise' pattern there: the reason is that over Max models, unsuccessful LDFS(s) calls do not necessarily result in an increase of the value of s. For example, if s admits a single action a, and $V(s) = Q_V(a, s) = \max[V(s_1), V(s_2)]$, then an increase in $V(s_2)$ does not lead to an increase in $V(s)$ if $V(s_2)$ remains lower than $V(s_1)$. This happens because the Max operator, unlike the Sum operator of Additive models, is not strictly monotone over all its arguments. A proposition analogous to Proposition 7, however, can be obtained for Max models provided that LDFS is extended with an extra $Bound$ parameter, resulting in a Bounded LDFS variant that reduces to IDA* with transposition tables even when the value function is not monotone. More interestingly, for Game Trees, Bounded LDFS reduces to the MTD$(-\infty)$ algorithm: an iterative Alpha-Beta search algorithm with Null Windows, that iteration after iteration, moves the evaluation window from $-\infty$ up til the correct Game Tree value is found (Plaat *et al.* 1996). This Bounded LDFS variant is formulated and compared with AO* over Max AND/OR graphs in (Bonet & Geffner 2005). In this paper we focus on a different extension of the basic LDFS procedure that yields the ability to handle additive models like MDPs where solutions can be cyclic.

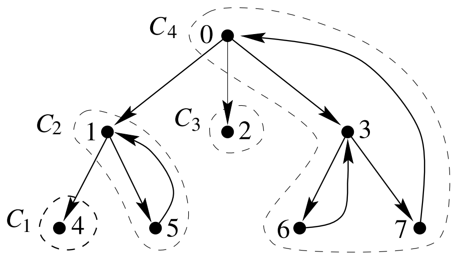

Figure 3: Labeling states in the presence of cyclic solutions in MDPs: the graph shows a cyclic policy π, where a link $i \rightarrow j$ means that state s_j is a possible successor of state s_i when action $\pi(s_i)$ is done in s_i. The states s_2 and s_4 are terminal states. LDFS(MDP) labels a state s as solved only when all the states s' that are reachable from s are consistent. For this, it uses Tarjan's bookkeeping mechanism for identifying the strongly-connected components C_i of the graph: when all the states in a component C_i are consistent, all of them are labeled as solved.

LDFS for MDPs

The LDFS algorithm solves all the models we have considered except MDPs. The reason is that in MDPs, there may be cyclic paths $s_i, a_i, s_{i+1}, a_{i+1}, \ldots, s_{i+k}, a_{i+k}, s_i$ with all the states along the path being consistent. This cannot happen in the other models. In such a case, LDFS would enter an infinite loop. For dealing with MDPs, however, it is not enough to detect and avoid such loops; the labeling mechanism that marks states as solved needs to be revised as well. In problems with cyclic solutions it is not enough to label a state s as solved when for some action a applicable in s, we have both $Q_V(a, s) \leq V(s)$ and all successors $s' \in F(a, s)$ solved. This is because the state s itself can be one of its own successors or a descendant of them. Also, a practical problem arising from the use of LDFS for MDPs has to do with the size of the residuals $\epsilon > 0$ allowed in the target value function. A value function V^* is optimal if its residuals $\min_{a \in A(s)} Q_V(a, s) - V(s)$, that are non-negative for a monotone V, are all zero. However, achieving zero residuals is costly and not strictly necessary: the number of updates for reducing the residuals to zero is not bounded, and non-zero residuals, if sufficiently small, do not to hurt optimality.

In light of these issues, LDFS(MDP) adds two features to the basic LDFS algorithm: an $\epsilon > 0$ bound on the size of the residuals allowed, and a bookkeeping mechanism for avoiding loops and recognizing states that are solved. To illustrate the subtleties involved in the latter task, let us assume that there is a single action $\pi(s)$ applicable in each state. By performing a single depth-first pass over the descendants of s, keeping track of visited states for not visiting them twice, we want to know when a state s can be labeled as solved. Due to the presence of cycles, it is not correct to label a state s as solved when the variable $flag$ indicates that all descendants of s are consistent (i.e., have residuals no greater than ϵ), as there may be ancestors of s that are reachable from s but have not been yet explored. Even in such cases, however, there must be states s in the DFS tree spanned by LDFS(MDP) (recall that no states are visited twice) such that the states that are reachable from s and π are all *beneath* s, and for those states, it is *correct* to label them as solved when $flag$ is true. Moreover, the labeling scheme is also *complete* if at that point not only s is labeled as solved but also its descendants. The question of course is how to recognize such 'top' elements in the state graph during the depth-first search. The answer is given by Tarjan's strongly connected components algorithm. Tarjan's algorithm (Tarjan 1972) identifies the strongly connected components of a directed graph by means of a depth-first traversal that keeps track of a pair of indices $s.low$ and $s.idx$ for each of the states visited. The indices are equal only for to the 'top' elements of each strongly connected component. For example, in the graph shown in Fig. 3, where a link $i \to j$ means that state s_j is a possible successor of state s_i (when action $\pi(s_i)$ is done in s_i), the components are C_1, C_2, C_3, and C_4. The LDFS(MDP) algorithm labels all the states s' in a component C as solved, when upon leaving the component, the value of the $flag$ variable indicates that no inconsistency has been found.

The resulting algorithm for MDPs, LDFS(MDP), is shown in Fig. 4. LDFS(MDP) is simply LDFS plus two additional features: non-zero ϵ-residuals and a labeling scheme based on Tarjan's strongly connected components algorithm. The correctness of LDFS(MDP) follows directly from the Find-and-Revise schema and the correctness of the Tarjan's algorithm:

Proposition 8 LDFS(MDP) *is an instance of* FIND*-and-*REVISE$[\epsilon]$ *and hence for a sufficiently small* ϵ*, solves MDPs provided they have a solution with finite (expected) cost.*

Since in every iteration FIND-and-REVISE$[\epsilon]$ either terminates or increases the value function of a greedy state by at least ϵ, the number of trials of both FIND-and-REVISE$[\epsilon]$ and LDFS(MDP) is bounded by $\sum_{s \in S}[\min(V^*(s), MaxV) - V(s)]/\epsilon$, where $MaxV$ stands for any finite upper bound on the optimal costs of the states that are not dead-ends. LDFS(MDP) works even in the presence of dead-ends s, provided that some policy exists that avoids such dead-ends and reaches the goal with probability 1. The ability to handle spaces that contains dead-ends is rather novel and does not extend to the standard algorithms, and algorithms like LRTA* and RTDP that presume that the goal is reachable from every state or, equivalently, that a proper policy exists.

The use of Tarjan's algorithm for labeling states in MDPs is borrowed from the HDP algorithm in (Bonet & Geffner 2003a). The two algorithms, however, are not equivalent. In particular, for deterministic actions, LDFS(MDP) reduces to IDA* with transposition tables, while HDP does not. Actually in such a case, HDP performs a sequence of 'greedy' searches where in every consistent state s starting with s_0 only *one* action a with $Q_V(a, s) = V(s)$ is considered. LDFS, on the other hand, like IDA* but unlike RTDP, considers all such actions in depth-first fashion. The result is that in certain cases, the number of iterations of HDP can be exponentially larger than in LDFS(MDP).

Actually, LDFS, HDP, and RTDP can all be understood in terms of the choice of the subgraph considered in each it-

```
LDFS(MDP)-DRIVER($s_0$)
begin
    while $s0$ is not SOLVED do
        LDFS(MDP)($s_0, \epsilon, 0, stack$)
        Clear ACTIVE bit on all visited states
    return $(V, \pi)$
end

LDFS(MDP)($s, \epsilon, index, stack$)
begin
    if $s$ is SOLVED or terminal then
        if $s$ is terminal then $V(s) := c_T(s)$
        Mark $s$ as solved return $true$
    if $s$ is ACTIVE then return $false$                % Update
    Push $s$ into $stack$
    $s.idx := s.low := index$
    $index := index + 1$
    % $V(s) := min_{a \in A(s)} Q_V(a, s)$                          ⋆
    $flag := false$
    foreach $a \in A(s)$ do
        if $Q_V(a, s) - V(s) > \epsilon$ then continue
        Mark $s$ as ACTIVE
        $flag := true$
        foreach $s' \in F(a, s)$ do
            if $s'.idx = \infty$ then
                $flag :=$ LDFS($s', \epsilon, index, stack$) & $flag$
                $s.low := \min\{s.low, s'.low\}$
            else if $s'$ is ACTIVE then
                $s.low := \min\{s.low, s'.idx\}$
        % $flag := flag$ & $[Q_V(a, s) - V(s) \leq \epsilon]$          ⋆
        if $flag$ then break
        while $stack.top.idx > s.idx$ do
            $stack.top.idx := stack.top.low := \infty$
    if $\neg flag$ then
        $V(s) := \min_{a \in A(s)} Q_V(a, s)$          % Update
        $\pi(s) := a$
        $s.idx := s.low := \infty$
        Pop $stack$
    else if $s.low = s.idx$ then
        while $stack.top.idx \geq s.idx$ do
            Mark $s$ as SOLVED                     % Labeling
            $stack.top.idx := stack.top.low := \infty$
            Pop $stack$
    return $flag$
end
```

Figure 4: LDFS for MDPs. The commented lines marked with ⋆ are not part of LDFS(MDP) but of the LDFS+ variant discussed in the text.

eration of FIND-and-REVISE: namely, whether a single best action is considered in each state or all of them, whether a single successor state is considered after each action or all of them, and whether inconsistent states are regarded as tip nodes in the search or not. The choices in LDFS are motivated by the goal of accounting for existing state-of-the-art algorithms such that IDA* and MTD. The choices in RTDP and HDP are somehow more ad-hoc, although, as we will see in the next section, not necessarily less effective. What works best in one model does not necessarily work best in other models, although having a general picture will turn out to be useful too.

Experimental Results

We evaluate the performance of LDFS(MDP) by comparing it with Value Iteration and other heuristic-search algorithms for MDPs. The heuristic-search algorithms chosen for comparison are Labeled RTDP (Bonet & Geffner 2003b), HDP (Bonet & Geffner 2003a) and Improved LAO* (Hansen & Zilberstein 2001). Value Iteration is included as a useful baseline.

The experiments were performed on Linux machines running at 2.8GHz with 2Gb of memory with a time bound of 10 minutes per experiment. Unless otherwise said, the parameter ϵ is set to 10^{-4}.

The first domain considered is the deterministic, standard version of the well known 8-puzzle game. We tested the algorithms with the sum-of-manhattan-distances heuristic over an instance with $h(s_0) = 12$ and $V^*(s_0) = 20$. LDFS(MDP), which is exactly IDA* with a transposition table for this problem, takes 0.001 seconds to find an optimal path. The time for the other algorithms are 0.004 for HDP, 0.012 for ILAO, 0.117 for LRTDP, and 9.235 seconds for Value Iteration. We then tried a noisy version of the 8-puzzle where each operator works as intended with probability $p = 0.9$ but has no effect with probability $1 - p$. This is a simple MDP where the only loops in the optimal policy are self-loops. For the resulting problem, the times for LDFS(MDP) degrade a lot. Indeed, LDFS(MDP) takes then 0.295 seconds, becoming almost 300 times slower than in the deterministic case. On the other hand, LRTDP and ILAO run 4 times slower then, and HDP 100 times slower. See Table 1 for more details.

The performance degradation in LDFS(MDP) when noise is added to the problem is related to a well known shortcoming of IDA* algorithms that arises when action costs are real numbers rather than integers (Wah & Shang 1994). In such a case, each iteration of IDA* may end up exploring very few new nodes so that an exponential number of iterations may be needed for convergence. Indeed, while LDFS(MDP) requires 9 iterations and 590 updates to converge in the deterministic setting, it requires $13,261$ iterations and $117,204$ updates when p is changed from 1 to $p = 0.9$. While this problem does not have an accepted solution, we have found that much better results can be obtained if two small changes are introduced in LDFS(MDP). These changes, that do not affect the correctness of the algorithm, correspond to adding the two lines of code that are marked with a $\star$ and commented in Fig. 4. The first line, $V(s) := \min_{a \in A(s)} Q_V(a,s)$, is an update that makes the state s consistent before the recursion, so that s is always explored. A consequence of this modification is that the condition $flag = true$ upon return from the recursion no longer guarantees that s is consistent *after* the recursion. Revising the value of $flag$ to reflect this condition is the role of the second line of code added to LDFS(MDP): $flag := flag \,\&\, [Q_V(a,s) - V(s) \leq \epsilon]$. These two changes make the searches deeper, partially overcoming the problem arising in IDA* in the presence of real edge-costs, resulting in general in a better performance. We will refer to version of the LDFS(MDP) algorithm with these two additional lines of code, as LDFS+. Table 1 shows the performance of LDFS+ over the 8-puzzle. As it can be seen, LDFS+ does not do as well as LDFS(MDP) in the deterministic case (where it comes second right after LDFS(MDP)), but improves over LDFS(MDP) in the stochastic setting (where it comes second right after ILAO).

The second domain considered in the evaluation is the racetrack benchmark introduced in (Barto, Bradtke, & Singh 1995) and used since then in a number of works. An instance in this domain is characterized by a racetrack divided into cells, and the task is to find the control for driving a car from a set of initial states to a set of goal states, minimizing the number of time steps. The states are tuples (x, y, dx, dy) that represent the position and speed of the car in the x, y dimensions, and the actions are pairs $a = (ax, ay)$ of instantaneous accelerations where $ax, ay \in \{-1, 0, 1\}$. Uncertainty comes from assuming that the road is 'slippery' and as a result, the car may fail to change acceleration with probability $1-p$ regardless of the action taken. When the car hits a wall, its velocity is set to zero but its position is left intact (this is different than in (Barto, Bradtke, & Singh 1995) where the car is moved to the start position).

We tried the various algorithms over the two instances in (Barto, Bradtke, & Singh 1995), the larger instance in (Hansen & Zilberstein 2001), and 6 ring- and 4 square-shaped tracks of our own. The algorithms are evaluated with the heuristics $h = 0$ and $h_{\textit{min-min}}$; the latter reflecting the cost of a relaxation where non-deterministic effects are deemed as controllable (Bonet & Geffner 2003b).

The results for the first set of instances are shown in Table 2. The runtimes do not include the time to compute the heuristic values as we are interested in evaluating how well the various algorithms exploit the heuristic information, rather than in evaluating heuristics or their computation. As it can be seen from Table 2, LDFS+ dominates all algorithms on these instances with both heuristics, except for `ring-1` with $h = 0$ where LRTDP is best. The situation is slightly different on the square racetracks, shown in Fig. 5, where LDFS+ is beaten closely by HDP and LRTDP.

The third domain considered for evaluation is a square navigation grid in which some cells are wet and thus slippery. The number of wet cells is controlled with a parameter p such that each cell is chosen independently as wet with probability p. The amount of water on a cell determines the effects of the actions and their probabilities. In our setting, a wet cell can have two levels of water which are selected with

8-puzzle	VI		LRTDP		ILAO		HDP		LDFS(MDP)		LDFS+	
	time	updates	time	updates	time	updates	time	updates	time	updates	time	updates
$p = 1.0$	9.236	3,991,680	0.117	51,395	0.012	4,146	0.004	1,985	0.001	590	0.004	1,492
$p = 0.9$	14.733	6,168,960	0.449	169,960	0.046	9,373	0.427	156,049	0.295	117,204	0.096	50,317

Table 1: Comparison over standard and noisy version of the 8-puzzle where each operator achieves its intended effect with probability p and has no effect with probability $1 - p$. In both cases, the sum-of-manhattan-distances heuristic is used, while for $p < 1$, the bound on residuals is $\epsilon = 10^{-4}$. For $p = 1$, the optimal cost and heuristics for the initial state are $V^*(s_0) = 20$ and $h(s_0) = 12$.

algorithm	`small`	`big`	`bigger`	`ring-1`	`ring-2`	`ring-3`	`ring-4`	`ring-5`	`ring-6`
$\|S\|$	9,394	22,532	51,941	429	1,301	5,949	33,243	94,396	352,150
$V^*(s_0)$	14.459	26.134	50.570	7.498	10.636	13.093	18.530	24.949	31.142
$h_{\textit{min-min}}(s_0)$	11	18	37	6	9	11	15	20	25
VI($h_{\textit{min-min}}$)	1.080	3.824	14.761	0.022	0.105	0.611	5.198	23.168	197.964
LRTDP($h_{\textit{min-min}}$)	0.369	3.169	12.492	0.006	0.027	0.138	2.173	15.361	243.130
ILAO($h_{\textit{min-min}}$)	0.813	4.739	20.190	0.008	0.034	0.463	11.428	37.598	—
HDP($h_{\textit{min-min}}$)	0.468	5.357	30.174	0.007	0.034	0.180	2.159	11.473	153.150
LDFS+($h_{\textit{min-min}}$)	**0.196**	**1.077**	**4.542**	**0.003**	**0.014**	**0.083**	**1.022**	**4.892**	**80.068**
VI($h = 0$)	1.501	5.289	21.701	0.027	0.124	0.774	7.281	34.501	354.917
LRTDP($h = 0$)	0.880	6.232	29.836	**0.012**	0.109	0.356	6.005	171.829	—
ILAO($h = 0$)	2.430	14.200	54.208	0.024	0.109	0.908	11.863	71.103	—
HDP($h = 0$)	2.440	30.955	174.698	0.032	0.149	0.927	11.957	96.398	—
LDFS+($h = 0$)	**0.792**	**3.417**	**16.080**	0.013	**0.057**	**0.353**	**4.390**	**24.732**	**310.019**

Table 2: Data for various racetrack instances and convergence times in seconds for the different algorithms with the heuristics $h = 0$ and $h_{\textit{min-min}}$. Results are for $\epsilon = 10^{-4}$ and $p = 0.7$. Faster times are shown in bold. A dash means the algorithm didn't finish within the 10 minutes time bound.

uniform probability once the cell has been chosen as wet. The cells that aren't wet have zero level of water. There are four operators to move along the four axis of the grid. On dry cells, the operators have deterministic effects, while on wet cells the level of non-determinism depends on the level of water. In our case, we deal with non-deterministic operators that can result in up to 4 different successor states. We tried grids of different sizes with $p = 0.4$. For each size of the grid, we tried a number of random instances with different initial and goal positions. The curves in Fig. 6 display the average times until convergence. Roughly, we see that for $h = 0$ the algorithms can be ordered from best to worst as ILAO, LRTDP, LDFS+, VI and HDP, while for $h = h_{\textit{min-min}}$ as ILAO, LRTDP and LDFS+, HDP and VI. Interestingly, ILAO, that didn't do well in the racetracks, does pretty well in this domain. The plots shown are logscale, so small differences in the plot may indicate a large difference on runtimes.

The last domain involves a navigation task on a complete binary tree of depth n extended with loops. The initial state of the problem is the root node of the tree and the goal nodes are the 2^n leaves of the tree at depth n. There are two probabilistic actions, 'left' and 'right', for moving to the left and right son of a node. Such actions achieve their intended effect with a probability p_s that depends on the state s and fail with probability $1-p_s$. Failure here means going back to the parent node. In addition, some of the nodes in the tree are labeled as 'noisy' with (independent) probability $r = 0.4$. In such nodes, the actions behave differently: both actions 'left' and 'right' move with probability $p_s/2$ to each of the two sons, and with probability $1-p_s$ to the parent node. The model is completed by specifying how the probability p_s depends on the node s where the actions are taken. Each node of the tree can be represented with a binary string of 0s and 1s that trace the *unique* path from the root node up to that node; e.g. the root node is associated with the empty string, the leftmost leaf of the tree with the string of n 0's, the rightmost leaf with the string of n 1's, etc. If we let $\#s$ denote the number of 1's in the string for s, then p_s is defined as $p^{\#s}$ where p is a fixed parameter of the problem. Thus, as we move towards the rightmost leaf of the tree, the operators become less reliable at an exponential rate; e.g. an operator at the father of the rightmost leaf will have its intended effect with probability p^{n-1}. We tried different tree depths with the parameters $p = 0.7$ and $r = 0.4$ (i.e., roughly 40% of nodes in the tree are noisy). The results are shown in Fig. 7. As it can be seen, LDFS+ and ILAO are the best algorithms in this domain with LDFS+ running a bit faster.

Summary and Conclusions

LDFS combines the benefits of a general dynamic programming formulation with the effectiveness of heuristic-search techniques in a simple piece of code that performs iterated depth-first searches enhanced with learning. LDFS reduces to well-known state-of-the-art algorithms over some models, but yields novel and effective approaches for other models like AND/OR graphs or MDPs.

The LDFS framework is useful both for understanding existing heuristic-search algorithms over various settings in a unified manner, and for devising new effective algorithms in other settings. In this paper, we considered also the formu-

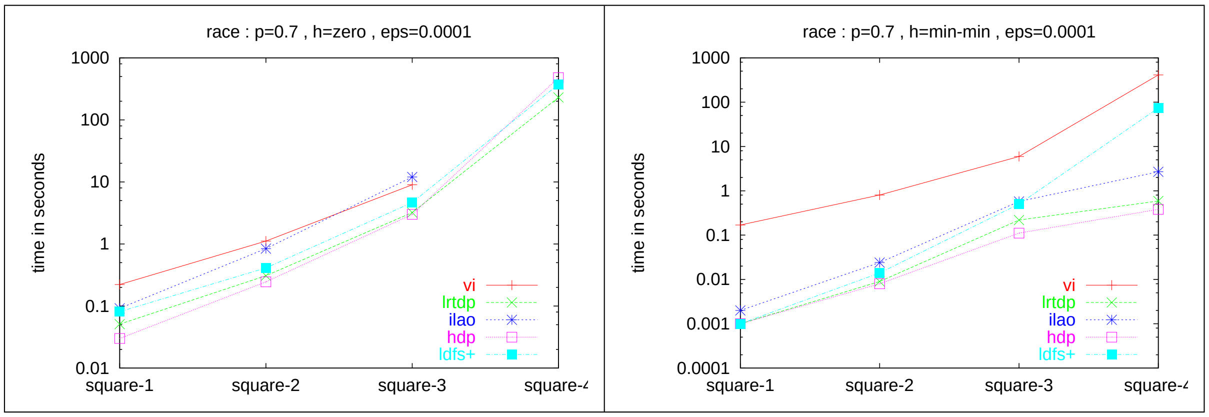

Figure 5: Average runtimes for convergence over the square racetracks for $p = 0.7$, $\epsilon = 10^{-4}$ and the heuristics $h = 0$ and $h_{\textit{min-min}}$.

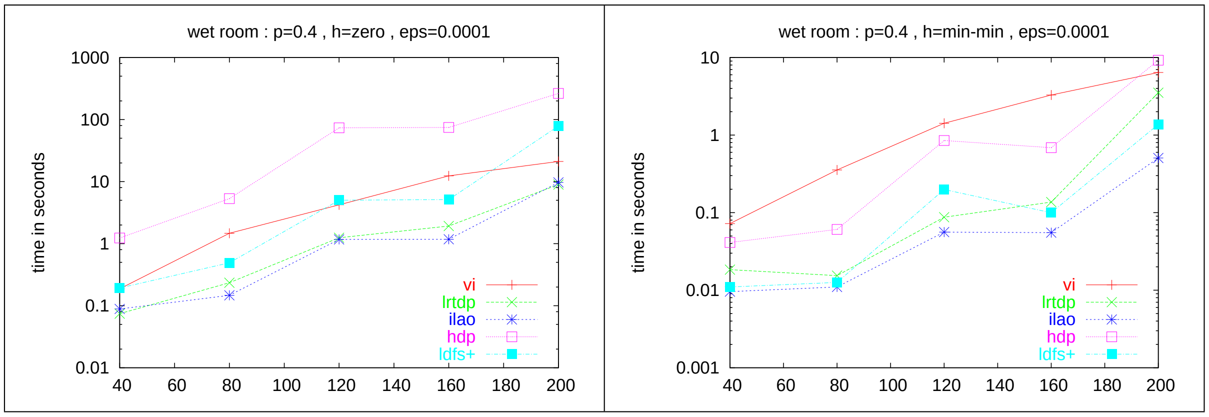

Figure 6: Average runtimes for convergence over the 'wet room' instances for $p = 0.4$, $\epsilon = 10^{-4}$, and the heuristics $h = 0$ and $h_{\textit{min-min}}$. The x-axis corresponds to the size of the room.

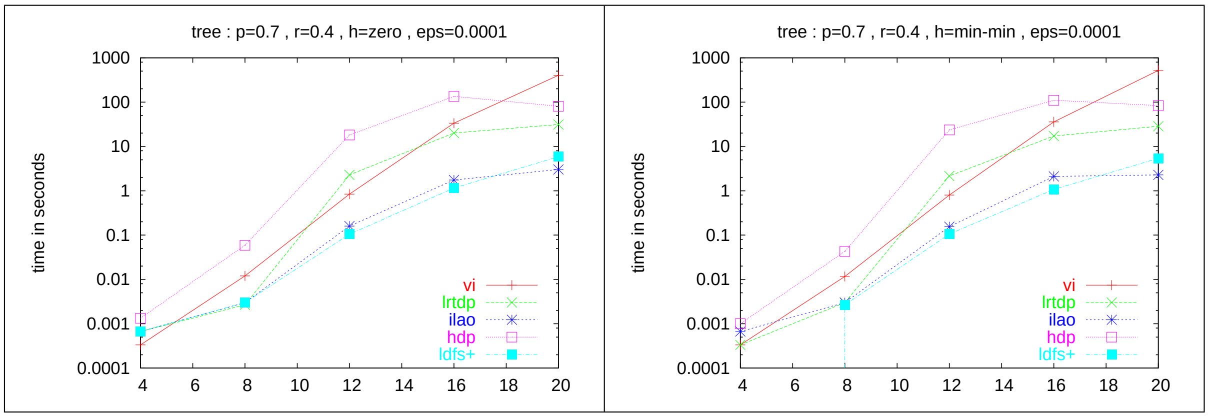

Figure 7: Average runtimes for convergence for navigation task over binary trees extended with loops, with $p = 0.7$, $r = 0.4$, $\epsilon = 10^{-4}$, and the heuristics $h = 0$ and $h_{\textit{min-min}}$. The x-axis corresponds to the depth of the tree.

lation of an LDFS algorithm for MDPs, and its evaluation in comparison with current state-of-the-art algorithms. Conceptually, LDFS shows that the key idea underlying several existing heuristic-search algorithms over various settings can be reduced to the use of iterated depth-first searches with memory for improving the heuristics over certain relevant states until they become optimal. From a practical point of view, the empirical results show that this idea leads to new algorithms that are competitive with current ones.

In order to improve LDFS further in the MDP setting, there are two concrete issues that we need to understand better. First, in (Bonet & Geffner 2005), it is mentioned that LDFS is much faster than AO* over Max AND/OR graphs, but not over Additive AND/OR graphs. Second, it is known that the performance of IDA* suffers when costs are real numbers and not integers, as the number of iterations can blow up. Both of these observations are relevant for solving MDPs that are additive models with real costs. The simple improvement of the LDFS for MDPs discussed in the previous section addresses these problems, but we expect that a deeper understanding of these issues may lead to further improvements. It is worth pointing out that these observations are not about MDPs per se, yet they are rendered relevant for MDPs thanks to the framework that connects these various models and algorithms.

Acknowledgements: We thank A. Frangi and A. Sanz for the use of the Hermes Computing Resource at the Aragon Inst. of Engr. Research (I3A), U. of Zaragoza. H. Geffner is partially supported by grant TIC2002-04470-C03-02 from MCyT/Spain and B. Bonet by grant from DID/USB/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. 1995. *Dynamic Programming and Optimal Control, (2 Vols)*. Athena Scientific.

Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In Gottlob, G., ed., *Proc. 18th International Joint Conf. on Artificial Intelligence*, 1233–1238. Acapulco, Mexico: Morgan Kaufmann.

Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In Giunchiglia, E.; Muscettola, N.; and Nau, D., eds., *Proc. 13th International Conf. on Automated Planning and Scheduling*, 12–21. Trento, Italy: AAAI Press.

Bonet, B., and Geffner, H. 2005. An algorithm better than AO*? In Veloso, M., and Kambhampati, S., eds., *Proc. 20 National Conf. on Artificial Intelligence*, 1343–1348. Pittsburgh, PA: AAAI Press / MIT Press.

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

Korf, R. 1985. Depth-first iterative-depeening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

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

Martelli, A., and Montanari, U. 1973. Additive AND/OR graphs. In Nilsson, N., ed., *Proc. 3rd International Joint Conf. on Artificial Intelligence*, 1–11. Palo Alto, CA: William Kaufmann.

McMahan, H. B.; Likhachev, M.; and Gordon, G. J. 2005. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In De Raedt, L., and Wrobel, S., eds., *Proc. 22nd International Conf. on Machine Learning*.

Newell, A.; Shaw, J. C.; and Simon, H. 1963. Chess-playing programs and the problem of complexity. In Feigenbaum, E., and Feldman, J., eds., *Computers and Thought*. McGraw Hill. 109–133.

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

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

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence* 87(1-2):255–293.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 16(7):701–710.

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

Wah, B., and Shang, Y. 1994. A comparative study of IDA*-style searches. In *Proc. 6th International Conf. on Tools with Artificial Intelligence*, 290–296. IEEE Computer Society.