Bootstrapping LPs in Value Iteration for Multi-Objective and Partially Observable MDPs

Diederik M. Roijers
Vrije Universiteit Brussel &
Vrije Universiteit Amsterdam

Erwin Walraven
Delft University of Technology
Delft, The Netherlands

Matthijs T. J. Spaan
Delft University of Technology
Delft, The Netherlands

Abstract

Iteratively solving a set of linear programs (LPs) is a common strategy for solving various decision-making problems in Artificial Intelligence, such as planning in multi-objective or partially observable Markov Decision Processes (MDPs). A prevalent feature is that the solutions to these LPs become increasingly similar as the solving algorithm converges, because the solution computed by the algorithm approaches the fixed point of a Bellman backup operator. In this paper, we propose to speed up the solving process of these LPs by bootstrapping based on similar LPs solved previously. We use these LPs to initialize a subset of relevant LP constraints, before iteratively generating the remaining constraints. The resulting algorithm is the first to consider such information sharing across iterations. We evaluate our approach on planning in Multi-Objective MDPs (MOMDPs) and Partially Observable MDPs (POMDPs), showing that it solves fewer LPs than the state of the art, which leads to a significant speed-up. Moreover, for MOMDPs we show that our method scales better in both the number of states and the number of objectives, which is vital for multi-objective planning.

Introduction

Several exact algorithms for solving a variety of decision-making problems in Artificial Intelligence, such as Multi-Objective Markov Decision Processes (MOMDPs; Barrett and Narayanan 2008), Partially Observable Markov Decision Processes (POMDPs; Kaelbling, Littman, and Cassandra 1998) and zero-sum Markov games (Littman 1994; 2001) rely on solving sets of linear programs (LPs). For example, the popular class of value iteration (VI) algorithms for decision-theoretic planning applies the Bellman backup operator until the fixed point is reached (Bellman 1957). If value functions are represented by sets of vectors, then the value iteration algorithm uses a pruning subroutine to remove dominated vectors. The LPs solved in this pruning subroutine become increasingly similar as the solution computed by the value iteration algorithm converges towards the fixed point, but so far this property has not been exploited and LPs are solved from scratch in every iteration. There is a significant amount of computation time to be saved here, because in exact VI algorithms LP solving takes up a large proportion of the total running time (Cassandra, Littman, and Zhang 1997).

As a concrete example we consider the incremental pruning algorithm for solving POMDPs. In the pruning subroutine of this algorithm a collection of LPs is solved, in which each constraint corresponds to a vector in the value function. Rather than solving these LPs immediately based on all constraints, it has been shown that it is more efficient to construct the LP incrementally by generating and adding constraints one by one (Walraven and Spaan 2017). This technique adds constraints that maximally reduce the LP's objective value, until an optimal LP solution is found. It first adds constraints at the extrema of the belief simplex, before approaching the belief point and constraints that constitute the optimal LP solution. Typically, constraints added in early stages of the algorithm turn out to be superfluous in hindsight.

Generating LP constraints incrementally has shown to be a powerful method to accelerate the vector pruning subroutine in incremental pruning for POMDPs (Walraven and Spaan 2017). In this paper we show that the iterative LP solution method for vector pruning can also be applied in the context of MOMDPs, rather than just POMDPs. However, in contrary to POMDPs, the superfluous constraints added by this method can lead to a significant number of iterations that can nullify the benefits of building up the LPs incrementally.

Our main contribution is the Bootstrap LP algorithm (BLP), which circumvents adding constraints that later turn out to be irrelevant. BLP bootstraps using similar LPs from the previous iteration of VI, as a heuristic to initialize LP constraints in a new iteration of VI. In particular, it takes a new LP in the current iteration of VI, identifies the most similar LP from the previous iteration and extracts the relevant constraints that defined the solution of this LP. Then, for every such constraint it identifies the most similar constraint in the current new LP. The BLP algorithm then starts to build the new LP by first adding all those constraints, before iteratively generating the remaining constraints. If the LP from the previous iteration is sufficiently identical, this leads to a reduction in the number of constraints added. To our knowledge BLP is the first algorithm that considers information sharing between LPs across different iterations in VI algorithms.

In our evaluation we show that incremental generation and bootstrapping of constraints leads to significant performance improvements in two state-of-the-art exact value iteration algorithms: CHVI for MOMDPs (Barrett and Narayanan 2008) and incremental pruning for POMDPs (Cassandra,

Littman, and Zhang 1997). We show that BLP needs fewer iterations in which constraints are added, leading to a significant speed-up while the additional overhead introduced by the bootstrapping method remains small. This leads to better scalability in both the number of states and the number of objectives in MOMDPs, which is key in MOMDP planning. Furthermore, we show that also in exact POMDP planning significant speed-ups can be achieved. From a more general perspective, our work and evaluation shows that LPs can be solved more efficiently by exploiting existing information from previous iterations, rather than solving each LP as a stand-alone problem.

Background

In this section we provide a general introduction to value iteration, MOMDPs, POMDPs and pruning of vectors.

MDPs and Value Iteration

A Markov Decision Process (MDP; Puterman 1994) consists of a finite set of states S, a set of actions A, a reward function R and a transition function T. When executing action $a \in A$ in the current state s, then the state transitions to $s' \in S$ based on the probability distribution $T(s,a,s') = P(s'|s,a)$, and the reward $R(s,a,s')$ is received. The goal is to maximize the expected discounted sum of reward $E[\sum_{t=0}^{\infty} \gamma^t R_t]$, where $0 \leq \gamma < 1$ is the discount rate and R_t is the reward at time t. A solution to an MDP consists of a policy $\pi : S \rightarrow A$, dictating which action should be executed in each state.

The quality of a policy can be expressed in terms of value functions. The value $V^{\pi}(s)$ denotes the expected discounted sum of reward when following policy π starting from state s. It is defined as $V^{\pi}(s) = E_{\pi}[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s]$. The value function V^* of an optimal policy π^* satisfies the Bellman optimality equation (Bellman 1957):

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V^*(s')). \quad (1)$$

The maximizing action a in the definition of $V^*(s)$ corresponds to the optimal action to be executed in state s.

Optimal value functions and hence optimal policies can be computed using value iteration algorithms. Value iteration initializes a value function V_0 and uses the Bellman optimality equation as an update rule to generate a new value function V_{n+1} based on value function V_n. This is also known as executing a backup operator H, such that $V_{n+1} = HV_n$, where H is defined by right hand side of Equation 1. Executing a sequence of backups yields a sequence of value functions, which is known to converge to a fixed point (i.e., $V_{n+1} = HV_n = V_n$). This occurs when the Bellman error magnitude, $\max_s |V_{n+1}(s) - V_n(s)|$, has become 0.

Partially Observable MDPs

A Partially Observable Markov Decision Process (POMDP; Kaelbling, Littman, and Cassandra 1998) is an extension of MDPs in which states cannot be observed directly. It augments the MDP model with an observation set O and observation function Ω. Rather than observing a state s' directly after executing action a, an observation $o \in O$ is received based on probability distribution $\Omega(a,s',o) = P(o|a,s')$.

Since an agent is unable to observe the state directly, it maintains a belief $b \in \Delta(S)$ over states, which is updated using Bayes' rule. $\Delta(S)$ denotes the continuous set of probability distributions over S, and is also called the belief simplex. A POMDP can be seen as a belief-state MDP, which is defined over beliefs rather than states. Hence, an agent makes decisions based on a policy $\pi : \Delta(S) \rightarrow A$, with the corresponding value function $V^{\pi}(b) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(b_t, \pi(b_t)) \mid b_0 = b\right]$, where $R(b_t, \pi(b_t)) = \sum_{s \in S} R(s, \pi(b_t)) b_t(s)$ and belief b_t is the belief at time t. This value function is similar to the MDP value function defined earlier, but it is defined over beliefs rather than states. In the definition POMDP rewards $R(s,a)$ are not dependent on the successor state s'. This assumption can be made without loss of generality, because a reward function $R(s,a,s')$ can be converted into a reward function $R(s,a)$ by computing a weighted average over all successor states s'.

In the finite-horizon setting value functions are piecewise linear and convex, and they can be defined using a set of vectors (Sondik 1971). We let V denote a set of vectors and $V(b)$ corresponds to the value of belief b, such that we can define $V(b) = \max_{\alpha \in V} \alpha \cdot b$. An optimal value function V^* can be computed using a series of backups. The value function V_0 can be initialized as $V_0(b) = \max_{a \in A} \sum_{s \in S} R(s,a) b(s) = \max_{a \in A} \; b \cdot \alpha_0^a$, where α_0^a is a vector such that the entry $\alpha_0^a(s)$ denotes the immediate reward $R(s,a)$, and the operator $\cdot$ denotes the inner product. Similar to MDPs, we can use the following Bellman backup operator H to generate V_{n+1} for a given value function V_n:

$$HV_n = \bigcup_{a \in A} G_a, \;\; \text{with } G_a = \bigoplus\nolimits_{o \in O} G_a^o \text{ and}$$
$$G_a^o \;=\; \left\{ \frac{1}{|O|} \alpha_0^a + \gamma g_{ao}^k \;\middle|\; 1 \leq k \leq |V_n| \right\}, \quad (2)$$

where the operator $\oplus$ denotes the cross-sum operator, which can be defined as $\mathcal{P} \oplus \mathcal{Q} = \{p + q \mid p \in \mathcal{P}, q \in \mathcal{Q}\}$ for two sets of vectors $\mathcal{P}$ and $\mathcal{Q}$. The vector g_{ao}^k is computed by creating a back-projection g_{ao}^k of the vector α_n^k from value function V_n using action a and observation o: $g_{ao}^k(s) = \sum_{s' \in S} P(o|a,s') P(s'|s,a) \alpha_n^k(s')$.

Because the value function $V_{n+1} = HV_n$ may contain dominated vectors, it is more efficient to compute the backup using $HV_n = \texttt{prune}\left(\bigcup_{a \in A} G_a \right)$, where

$$G_a = \texttt{prune}\left(\texttt{prune}\left(\bar{G}_a^1 \oplus \bar{G}_a^2 \right) \ldots \oplus \bar{G}_a^{|O|} \right), \quad (3)$$

and $\bar{G}_a^o = \texttt{prune}(G_a^o)$, following the incremental pruning scheme (Cassandra, Littman, and Zhang 1997).

Multi-Objective MDPs

In a Multi-Objective Markov Decision Process (MOMDP; Roijers et al. 2013), the reward function $R(s,a,s')$ is vector-valued rather than scalar. This enables the modeling of problems with two or more objectives, for which preferences between all possible trade-offs cannot be specified a priori.

Algorithm 1: Vector pruning (White & Lark)

input : vector set U
output : pruned set D (result after pruning U)

$D \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
  $u \leftarrow$ arbitrary element in U
  if $v \succeq u, \exists v \in D$ **then**
    $U \leftarrow U \setminus \{u\}$
  else
    $x \leftarrow$ `FindPoint(`D, u`)`
    if $x = \phi$ **then**
      $U \leftarrow U \setminus \{u\}$
    else
      $u \leftarrow$ `BestVector(`x, U`)`
      $D \leftarrow D \cup \{u\}$
      $U \leftarrow U \setminus \{u\}$
    end
  end
end
return D

In MOMDPs the values of policies are also vector-valued rather than scalar. However, as vectors permit only a partial ordering, the values of states, $V^*(s)$, become sets of value vectors (Barrett and Narayanan 2008). Specifically, these sets must contain an optimal value vector for every preference or utility function, f, that a user might have. We focus on the highly prevalent scenario (Roijers et al. 2013) in which the utility function is a linear function, i.e., $u_w = f(u, w) = w \cdot u$, in which u is a value vector and w is a vector specifying the relative importance of each objective. A set that has one optimal value vector for every possible w, is called a Convex Coverage Set (CCS) or Convex Hull (CH).

Convex Hull Value Iteration (CHVI) is an exact MOMDP planning algorithm (Barrett and Narayanan 2008) that iteratively applies an updated backup operator until convergence:

$$HV_n(s) = \texttt{prune}\left(\bigcup_{a \in A} \bigoplus_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V_n(s'))\right), \quad (4)$$

where $R(s,a,s')$ is a single vector, $V_n(s')$ is a set of vectors, and the $+$ operator translates all the vectors in $\gamma V_n(s')$ by $R(s,a,s')$. However, the resulting set can contain excess vectors, i.e., vectors that are not necessary to build a CH. Similar to POMDPs, such vectors are removed using a pruning operator `prune`. This operator removes excess value vectors w.r.t. optimality for all w. Note that this is the same operator as needed to remove excess vectors w.r.t. optimality for all b in POMDPs, and pruning can also executed incrementally as in the incremental pruning algorithm for POMDPs.

Pruning Vectors

Vector pruning works identically for both MOMDPs and POMDPs. Value functions of MOMDPs are a function of weight vectors w, and value functions of POMDPs are a function of beliefs b. In order to provide a general introduction to pruning, we refer to either of them as a point x.

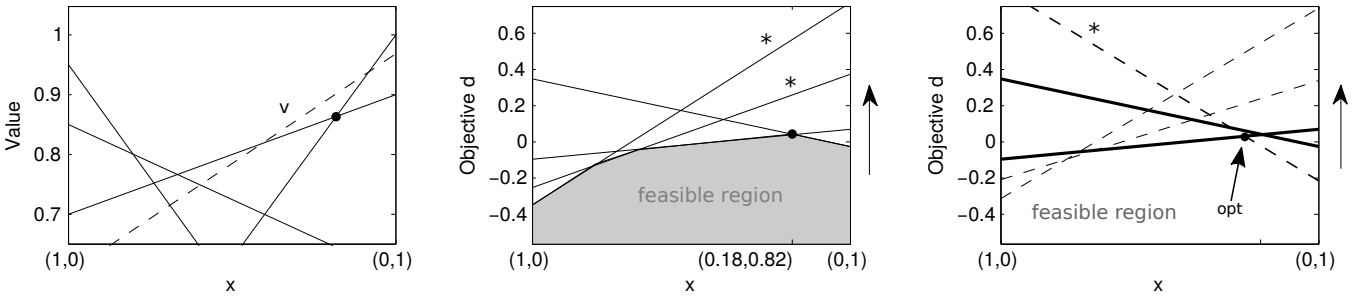
The subroutine `prune` can be implemented using a method proposed by White and Lark (White 1991), as shown in Algorithm 1. The symbol $\succeq$ denotes that vector v fully dominates u. The subroutine `BestVector` returns the vector in U with the maximum value in point x (Littman 1996). The subroutine `FindPoint`(U, v) uses linear programming to find a point at which value function U improves the most if vector v is added to U. An example is shown in Figure 1a, where the solid lines represent vectors $u \in U$ and the vector v is represented by the dashed line. It can be seen that adding vector v to U does improve the value function the most at the highlighted point. Figure 1b visualizes the feasible region of the corresponding LP, where each line corresponds to a constraint $d^* \leq (v - u) \cdot x$. The shaded area represents the feasible region of the LP, and the arrow indicates the direction of optimization. Hence, the corner represented by the dot corresponds to the optimal solution of the LP.

The LP in `FindPoint` can be solved directly based on all constraints, but it has been shown that this procedure can be implemented more efficiently by generating LP constraints incrementally (Walraven and Spaan 2017). This method is shown in Algorithm 2 and also determines whether adding vector v to the set U improves the value function induced by U. It generates the constraints incrementally until an optimal solution has been found. We refer to this method as Incremental Constraint Generation (ICG). An attractive property of ICG is that it does not necessarily generate all constraints. In other words, the algorithm may terminate before enumerating all constraints of the LP. An example is shown

Algorithm 2: `FindPoint(ICG)` – computes the point in which v improves U the most (Walraven and Spaan 2017)

input : vector set U, vector v
output : point x or symbol ϕ

if $|U| = 0$ **then**
  return arbitrary point x
end
$\ell \leftarrow$ length of vector v
define the following LP:
  max d^*
  s.t. $\sum_{i=1}^{\ell} x_i = 1$
    $x_i \geq 0 \qquad i = 1, \ldots, \ell$
    d^* free
choose an arbitrary point x'
do
  $\bar{x} \leftarrow x'$
  $\hat{u} \leftarrow \arg\min_{u \in U} \{(v - u) \cdot \bar{x}\}$
  add $d^* \leq (v - \hat{u}) \cdot x$ to the LP
  solve the LP to obtain point x'
while $x' \neq \bar{x}$;
$\bar{d} \leftarrow$ last objective d^* found
return $\bar{x}$ if $\bar{d} > 0$ and ϕ otherwise



(a) Value function U and vector v (b) Feasible region of the LP (c) Bootstrapping constraints

Figure 1: Value function U and vector v, the corresponding LP and a bootstrapping example

in Figure 1b, in which the constraints labeled by the stars do not intersect in the corner corresponding to the optimal solution. Such constraints are not strictly necessary to define an optimal LP solution, and they do not need to be enumerated. Prior to solving it is unknown whether a constraint needs to be enumerated, because this requires a priori knowledge about the optimal solution of the LP.

Bootstrapping LP Solutions

Previous approaches to make LP-based pruning faster rely on alternative LP formulations for pruning (Feng and Zilberstein 2004). Such formulations have had a significant positive impact on the number of the LPs or the speed with which they are solved. However, until now LPs have always been solved in isolation. Our key insight is that we can share information across multiple LPs in different iterations in VI. We observe that LP solutions get increasingly similar when value iteration algorithms converge, and we show how an initial subset of LP constraints can be initialized based on similar LPs from previous iterations of VI, such that Algorithm 2 does not need to start with an empty constraint set when generating constraints incrementally.

Analysis

In this section we consider two successive value functions V_{n-1} and V_n. Note that this differs slightly from the value function introduction in the background section, where we used V_n and V_{n+1}. An important observation about value iteration is that as the iterations in value iteration progress, the Bellman error magnitude (i.e., the maximum difference between two successive value functions) becomes smaller (Puterman 1994). In MDPs this is across all states, in MOMDPs across all states and weight vectors, and in POMDPs across all beliefs. Therefore, we expect the set of value vectors V_n in incremental pruning to contain increasingly similar vectors to V_{n-1}. To identify which vectors need to be retained in V_n, Algorithm 1 incrementally builds up this set. Starting from an empty set, this algorithm identifies a point, x, for which a candidate vector is possibly optimal. The best vector for that x is added to V_n. To identify x, Algorithm 2 solves a series of LPs with an increasing number of constraints.

Firstly, we observe that the points identified by Algorithm 2 become increasingly similar during the execution of value iteration. We consider a vector v_{n-1} in iteration $n-1$ and a vector v_n in iteration n. Both vectors are used as input to the ICG LP (Algorithm 2) as the candidate vector and if the vectors are similar then the identified point returned by Algorithm 2 will be similar too. In other words, similar vectors in two successive value functions will be optimal for similar beliefs. This is because VI converges to a fixed point of the value function, which is a convex set of vectors. In our observation we implicitly make the assumption that the pruning algorithm always considers the vectors in a specific (e.g., lexicographic) order, which we further discuss later.

Secondly, we make use of the observation that there are only a handful of constraints, $\mathcal{C}_{prev}$, that ultimately constitute the solution of the LP (Walraven and Spaan 2017), as illustrated in Figure 1b. These can be identified easily (following the notation of Algorithm 2) as:

$$\mathcal{C}_{prev} = \arg\min_{u \in U} (v - u) \cdot x^*, \tag{5}$$

where x^* is the ultimately returned point. Note that $\mathcal{C}_{prev}$ contains vectors corresponding to the constraints intersecting in x^*. For the example this would be the vectors intersecting in the dot in Figure 1a. Since there is a direct correspondence between vectors and constraints, we will use both terms interchangeably if the meaning is clear from context.

By combining the two observations we hypothesize that we can reuse $\mathcal{C}_{prev}$ for other vectors similar to vector v. We consider iteration $n-1$, in which `FindPoint(ICG)` was called with a vector set U_{n-1} and a vector v_{n-1}. If the same function is called in iteration n with vector set U_n and a similar vector v_n, then we select the closest vectors from the new set U_n (according to Euclidean distance) to initialize the LP in iteration n:

$$\mathcal{C}_{init} = \bigcup_{v_{n-1} \in \mathcal{C}_{prev}} \arg\min_{v_n \in U_n} |v_n - v_{n-1}|. \tag{6}$$

This set contains vectors from U_n similar to the vectors from U_{n-1} which correspond to the constraints defining the optimal LP solution for v_{n-1}.

As an example we consider the LP shown in Figure 1b. Suppose that ICG solves this LP and finds the solution indicated by the dot. Now suppose that we encounter a similar LP in a subsequent iteration of incremental pruning, as shown in

Algorithm 3: `FindPoint(BLP)` – computes the point in which v improves U the most

input : vector set U, vector v, and a context iteration number n, and context element θ
output : point x or symbol ϕ

1 **if** $|U| = 0$ **then**
2 | **return** arbitrary point x
3 **end**
4 $\ell \leftarrow$ length of vector v
5 $v_{n-1}, \mathcal{C}_{n-1}, x' \leftarrow \underset{(v',\mathcal{C},x') \in \texttt{cache}(n-1,\theta)}{\arg\min} |v - v'|$
6 $\mathcal{C}_{init} \leftarrow \bigcup_{c \in \mathcal{C}_{n-1}} \underset{u \in U}{\arg\min} |c - u|$
7 define the following LP:
8 max d^*
9 s.t. $\sum_{i=1}^{\ell} x_i = 1, \quad x_i \geq 0 \quad \forall i = 1, \ldots, \ell,$
10 $d^* \leq (v - \hat{u}) \cdot x \qquad \forall \hat{u} \in \mathcal{C}_{init}$
11 **do**
12 | $\bar{x} \leftarrow x'$
13 | $\hat{u} \leftarrow \arg\min_{u \in U} \{(v - u) \cdot \bar{x}\}$
14 | add $d^* \leq (v - \hat{u}) \cdot x$ to the LP
15 | solve the LP to obtain point x'
16 **while** $x' \neq \bar{x}$;
17 $\mathcal{C}_{prev} \leftarrow \arg\min_{u \in U} (v - u) \cdot \bar{x}$
18 add $(v, \mathcal{C}_{prev}, \bar{x})$ to the $\texttt{cache}(n, \theta)$
19 $\bar{d} \leftarrow$ last objective d^* found
20 **return** $\bar{x}$ if $\bar{d} > 0$ and ϕ otherwise

Figure 1c. In the figure each line corresponds to a constraint, and the feasible region is the area below the lines. In this case we would like to initialize constraints which are likely to be intersecting in the optimal LP solution. Therefore our bootstrapping technique initializes the LPs with the constraints shown as a bold solid line, as these constraints are similar to the constraints intersecting in the optimal solution in the previous iteration (see Figure 1b). In the new LP the optimal solution (indicated by opt) is slightly different compared to the previous LP, and BLP needs to add only one more constraint (labeled $*$). This is beneficial, as ICG would start from an empty LP, and iterates multiple times before reaching the same solution.

It is important to note that initialization of constraints based on $\mathcal{C}_{init}$ never shrinks the feasible region of the LP too much. The reason is that the constraints defined by $\mathcal{C}_{init}$ correspond to vectors from the current vector set U_n, rather than vectors from a previously solved LP. In other words: the algorithm always initializes constraints that are valid constraints in the LP that is currently being solved.

Vector Pruning Algorithm

To perform bootstrapping, we need to store and retrieve $\mathcal{C}_{prev}$. Furthermore, we observe that constraints and solutions that can be reused are context-dependent. For MOMDPs, this context is a state transition (s, a, s'). For POMDPs, the context is the (a, o)-pair of the $\bar{G}_a^o$ set that has just been added to the cross-sum in Equation 3. To integrate context-dependent bootstrapping we make the following changes, leading to a new algorithm that we call BLP.

First, the `prune` subroutine is implemented as an adapted version of Algorithm 1, with the following modifications: it is now parameterized by n, and context element θ (the transition (s, a, s') for MOMDPs and an action-observation-pair (a, o) for POMDPs), which it passes to our new subroutine to identify points while reusing LP information, `FindPoint(BLP)`, as implemented in Algorithm 3 (replacing Algorithm 2). This algorithm is described in the next section. On the top level, i.e., the `prune` call after the union over all sets, we use s in as the context in MOMDPs, and `null` in POMDPs.

Second, we ensure that the vectors that need to be pruned are lexicographically ordered. That is, each time Algorithm 1 is called on a set of vectors, U, we sort the set. Consistency in this sorting is crucial for our first observation in the previous section, as the order in which Algorithm 1 considers the vectors, influences the sequence of arguments `FindPoint` (U and v) is called with.

Point-Finding Algorithm

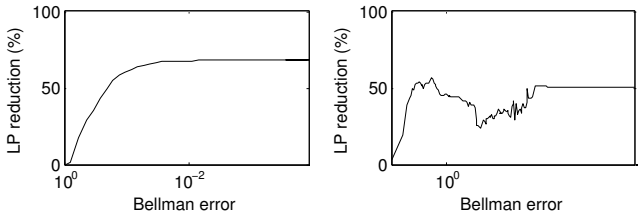
We now describe how bootstrapping is integrated in the original point-finding algorithm used by the pruning algorithm. The key improvement of our BLP algorithm over ICG is how it solves LPs by bootstrapping off the LPs from previous iterations, as specified in Algorithm 3.

First, BLP retrieves the relevant constraints from the previous iteration ($n - 1$). It does this by matching the closest vector, v' from the cache with the same context, (a,o) or (s,a,s'), from the previous iteration on line 5. Aside from v', the constraints $\mathcal{C}_{n-1}$ in the form of vectors from the previous iteration and the point that was optimal for the corresponding LP are also retrieved. If this is the first iteration, i.e., the cache is empty, we use a vector of zeroes, an empty set of constraints and a random point as default. Because $\mathcal{C}_{n-1}$ is in the form of vectors we can match the closest vectors from U in the current iteration, n, on line 6. These vectors are stored in a set, $\mathcal{C}_{init}$, and used to the initialize the constraints of the LP on line 10. The initial constraints are of the form: $d^* \leq (v - \hat{u}) \cdot x$, where v is the input vector, and $\hat{u}$ is a vector in $\mathcal{C}_{init}$. After constructing the initial LP, BLP generates the constraints incrementally until an optimal solution has been found, as in `FindPoint(ICG)` (Algorithm 2) on lines 11–16. This leads to the final solution $\bar{x}$ of the LP.

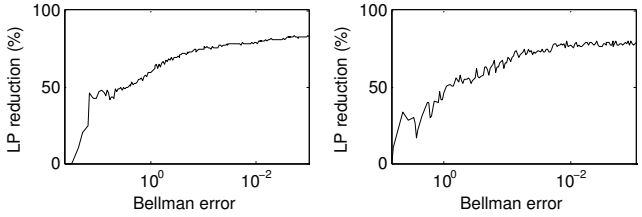
Given the LP solution $\bar{x}$, the method `FindPoint(BLP)` retrieves and stores constraints and $\bar{x}$ itself, for reuse in subsequent iterations. The constraints, $\mathcal{C}_{prev}$ are those $u \in U$ that are optimal for $\bar{x}$ (line 17). This is stored in the cache (line 18) before returning $\bar{x}$ if there is a point for which v is an improvement over U, or ϕ if there is not.

Discussion

BLP introduces overhead in the form of bookkeeping necessary to match the contexts and the similar constraints. Furthermore, sorting induces extra work: sorting is $O(|S||U| \log |U|)$

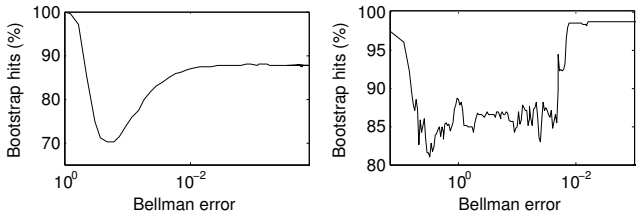


(a) 45-state-2-obj. R-MOMDP (b) G-DST (MOMDP)

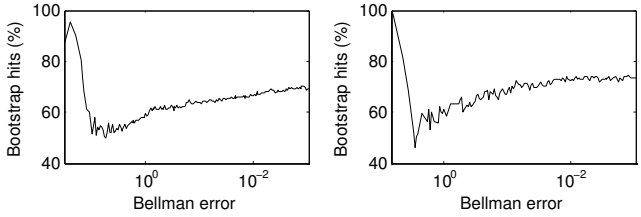


(c) Network (POMDP) (d) Shuttle (POMDP)

Figure 2: Reduction of the number of LPs as a function of the residual Bellman error.



(a) 45-state-2-obj. R-MOMDP (b) G-DST (MOMDP)



(c) Network (POMDP) (d) Shuttle (POMDP)

Figure 3: Average percentage of constraints successfully added due to bootstrapping (as function of the Bellman error).

for POMDPs and $O(c \cdot |U| \log |U|)$ for MOMDPs, where c is the number of objectives. Hence, it is not clear a priori that it will always be faster than ICG, or even White and Lark's pruning. However, we expect that our method will get increasingly better as the number of iterations performed increases, i.e., as the magnitude of the Bellman error goes down. We show this empirically in the next section. Finally, it should be noted that BLP does not change the solutions computed by the MOMDP and POMDP algorithms, since the calls to `FindPoint` and the corresponding output remain identical.

Experiments

In this section we present the results of our experimental evaluation for both MOMDPs and POMDPs.

Problem Domains

For all POMDP experiments we use benchmark domains from `pomdp.org`, which provides several standard domains that are typically used to evaluate POMDP algorithms.

For MOMDPs we first consider randomly generated MOMDPs with limited underlying structure as specified in the MORL-Glue benchmark suite (Vamplew et al. 2017). A transition matrix $T(s, a, s')$ is generated using $N = 3$ possible successor states per action, with random probabilities drawn from a uniform distribution. There are $|A| = 3$ actions, and a varying number of objectives. To ensure that every state is reachable from every state, it is enforced that for every state with a number x, $x{+}1 \mod |S|$ is one of the successor states for one of the actions.

The second domain we consider is Generalized Deep Sea Treasure (G-DST) (Vamplew et al. 2017), which is a generalisation of the popular grid-shaped deep sea treasure MOMDP benchmark (Vamplew et al. 2011). In the G-DST benchmark a submarine receives a reward for reaching a treasure on the bottom of the sea (objective 1), while receiving a -1 fuel reward (objective 2) for every move. The treasures have different values and are placed at various depths on the bottom of the sea. There are four actions (up, down, left, and right), which move in the corresponding direction in the grid, with probability $p = 0.8$ and again in a random direction with probability $1 - p = 0.2$. The agent starts in the top-leftmost square in the grid. The number of states in G-DST can be changed by adjusting the number of columns in the grid, with the restriction that a column is as deep or deeper than the column to its left (i.e., has more more vertical positions).

Number of LPs solved

In our discussion in the previous section we observed that BLP introduces additional overhead due to additional bookkeeping, and that we expect that our method performs better as the Bellman error becomes smaller. In this section we experimentally confirm this claim. We execute CHVI (for MOMDPs) and Incremental Pruning (for POMDPs) combined with BLP and measure the reduction in the number of LPs in each iteration of incremental pruning and CHVI (by comparing with ICG). As the Bellman error magnitude becomes smaller during these iterations, we can derive a relationship between the Bellman error and the reduction of the number of LPs.

Figure 2 shows the LP reduction realized by BLP as a function of the Bellman error for several domains. As the Bellman error goes down during the execution of incremental pruning, we reversed the x-axis and we use a log scale. A reduction of 40 percent means that BLP solves 40 percent fewer LPs compared to ICG in the same iteration. The experiment confirms our claim that the performance of BLP improves when the Bellman error becomes smaller and it confirms that BLP leads to a major reduction of the total number of LPs solved. Note that the reduction will never reach 100 percent, as this would correspond to eliminating all LPs, which is impossible. In our experiments the overhead of bookkeeping is typically around 5% of the total running time.

	ICG		BLP	
Domain	Time (s)	#LPs	Time (s)	#LPs
Tiger	2.6	41k	**2.1**	23k
Marking	3.9	37k	**3.5**	26k
Partpainting	4.3	57k	**3.7**	41k
Marking2	4.5	43k	**3.9**	30k
Stand.Tiger	251.0	3595k	**203.6**	1944k
Shuttle	430.0	3108k	**380.8**	2251k
Network	692.2	6038k	**595.4**	4018k
4x5x2	779.6	1326k	**769.6**	952k

Table 1: Comparison ICG and BLP on various POMDPs

	WL	ICG		BLP	
Domain	Time (s)	Time (s)	#LPs	Time (s)	#LPs
5s 2a 2o	3.2	3.7	35k	**3.1**	22k
5s 2a 2o	**4.6**	6.0	62k	4.7	33k
5s 2a 3o	154.6	127.2	1763k	**101.3**	817k
5s 2a 3o	307.4	263.7	3604k	**191.2**	1442k
5s 2a 4o	143.8	131.8	1775k	**120.2**	1171k
5s 2a 4o	1687.3	940.9	13070k	**867.4**	8293k
7s 3a 2o	**13.3**	18.3	285k	13.7	125k
7s 3a 2o	28.2	35.5	434k	**24.9**	169k
7s 3a 3o	1066.8	833.2	10608k	**619.4**	5096k
7s 3a 3o	1637.0	1068.7	15010k	**814.2**	6437k

Table 2: Comparison WL, ICG and BLP, on Random MOMDPs with varying numbers of states (s) actions (a) and objectives (o). Instances with the same number of states, actions and objectives are generated with different seeds.

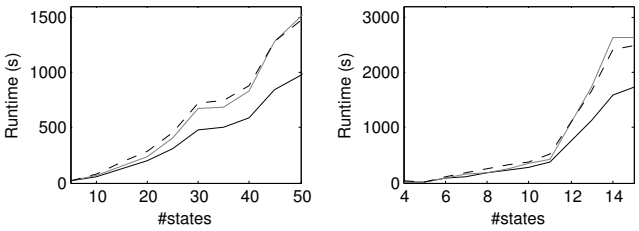


Figure 4: Runtime as a function of the number of states, for (left) 2-objective random MOMDPs with 3 actions, 3 possible successor states, (right) randomly drawn G-DST instances with a variable number of states. Lines indicate CHVI+WL (grey), CHVI+ICG (dashed) and CHVI+BLP (solid).

Bootstrapping Performance

Next, we study whether BLP adds relevant constraints based on the information from the previous iteration. To be more precise, we study whether Algorithm 3 adds constraints on line 10 based on $\mathcal{C}_{init}$, which would have been added iteratively by Algorithm 2 as well. If this is the case, it means that BLP successfully uses information from a previous iteration to initialize LPs in a new iteration. We execute Algorithm 3 (BLP) and Algorithm 2 (ICG) in parallel to measure the percentage of constraints BLP adds based on bootstrapping that are also added by ICG. We refer to this metric as *bootstrap hits*, which we report in Figure 3 as an average over the LPs solved in an iteration of value iteration.

We observe that our bootstrapping strategy adds more constraints that ICG would also add when the Bellman error becomes smaller. Intuitively, this can be explained by the observation that value functions and hence LPs become more similar when VI algorithms converge. The relatively large proportion of hits at the start of the curves can be explained by observing that there are few vectors during the first few iterations, and thus only a few possible LP constraints. For the results in Figure 3 we found that the standard deviation of the hits is typically around 20 percent. This may seem relatively large, but this can be explained by observing that, for example, perfect hits in small-sized LPs correspond to 100 percent, which slightly increases the standard deviation. We conclude that, confirming our expectations, the performance bootstrapping becomes better when the Bellman error decreases, and BLP successfully uses information from previous iterations to accelerate LP solving in subsequent iterations.

Runtime Performance

Now that we have tested how effective BLP is in reducing the amount of LPs compared to ICG, we test how the runtime and the number of solved LPs of BLP compare to ICG as well as White and Lark's algorithm (Algorithm 1), which we refer to as WL. We test both on POMDPs and on MOMDPs.

First, we compare incremental pruning for POMDPs combined with ICG (Algorithm 2) and incremental pruning combined with BLP (Algorithm 3) on the same benchmarks as Walraven and Spaan (2017). For clarity we use ICG and BLP to refer to the respective methods. We do not compare with WL in this experiment. We kindly refer to Walraven and Spaan (2017) for this comparison. We execute incremental pruning until the Bellman error drops below 0.02, and the reported running times are an average based on 10 independent runs. Due to the size of the domains we used an error tolerance of 0.05 for the Standing Tiger domain, and in the 4x5x2 domain we could only execute 14 iterations. The results are shown in Table 1, which reports the running times and the total number of LPs solved. We observe that reusing information about LPs from previous iterations of value iteration leads to a reduction of the running time, and a significant reduction of the running time in several domains.

Second, we compare the runtime of BLP to WL and ICG as a function of the size of the state space for Random 2-objective MOMDPs (Figure 4 left), and for G-DST instances (Figure 4 right), which also have 2 objectives. For both 2-objective Random MOMDPs and G-DST it is apparent that ICG does not improve the runtime compared to WL. In other words, just iteratively adding constraints does not lead to more effective pruning. BLP on the other hand effectively reuses the constraints between iterations, leading to significant speed-ups; for 2-objective Random MOMDPs with 10 states CHVI with BLP uses 91 percent of the runtime of CHVI with WL (45s versus 51s), at 35 states only 72 percent (499s versus 690s), and at 50 states only 63 percent (1014s versus 1606s). That BLP is significantly faster than ICG can

be explained by the effectiveness of reuse, at 10 states BLP needs only solve 42 percent of the number of LPs that ICG does ($354k$ versus $840k$), and at 50 states only 33 percent (7.4 mln versus 22.5 mln). We thus conclude that CHVI with BLP scales much better in the number of states than CHVI with WL, while CHVI with ICG does not.

That ICG does not improve the runtime for 2-objective MOMDPs is an interesting observation; for POMDPs Walraven and Spaan (2017) show that this is extremely effective. However, we note that in MOMDPs the total number of LPs is much larger than in POMDPs (compare, e.g., the numbers of LPs for 50-state MOMDPs—BLP: 7.4 mln, ICG: 22.5 mln—with POMDPs in Table 1) while the value vectors are much shorter in these MOMDPs (i.e., length 2) than in POMDPs (i.e., the number of states in the POMDP).

To compare the runtime of WL, ICG and BLP for an increasing number of objectives, we test the algorithms on several individual instances in Table 2, with a small number of states and actions (i.e., 5 and 2, and 7 and 3). Note that instances with the same size have been generated with a different seed. The results show that for the 2-objective problems, ICG is significantly slower than WL and BLP, and WL is still on par with BLP, as we also show in Figure 4. This is because for 5 and 7 states and only two objectives, the overhead needed for adding the LP constraints sequentially is still relatively high. BLP reduces this overhead significantly with respect to ICG, but not enough to make it significantly faster than WL. When the number of objectives is larger than two, ICG becomes faster than WL. However, BLP scales best in the number of objectives, and can almost halve the runtime w.r.t. WL in the two most difficult instances we tested.

In conclusion, for POMDPs, BLP is faster than the state-of-the-art ICG in all instances we tested, and much faster in some instances. For MOMDPs, BLP is much faster than WL and ICG, and scales better in both the number of states and the number of objectives, making this method a key improvement to MOMDP planning.

Related Work

Our work is related to region-based pruning (Feng and Zilberstein 2004), which uses different LPs to detect vector dominance, and exploits information about the cross sum when creating these LPs. The number of constraints in these LPs is polynomial in the size of the vector sets, rather than exponential in the worst case. In contrast to our work, the number of LPs remains the same. Another related pruning approach is Skyline (Raphael and Shani 2012), which traces the surface of the value function. ICG outperforms both region-based pruning and Skyline (Walraven and Spaan 2017).

Incremental construction of constraint sets has also been used in the approximate POMDP algorithm α-min (Dujardin, Dietterich, and Chadès 2015). It uses a mixed-integer problem in which so-called facets are generated incrementally, which resembles constraint generation. ICG and BLP select constraints from a known set of constraints, whereas the facets in α-min are used to approximate a set of constraints that is initially unknown. The latter is computationally more difficult, and both ICG and BLP do not need to rely on such a procedure since the constraint set is finite and already known.

Our work is related to decomposition approaches for linear programs, such as row and column generation (Benders 1962; Gilmore and Gomory 1961). Rather than solving an LP directly, such approaches decompose an LP into smaller parts to improve tractability of solving. Algorithm 2 has been derived using such a decomposition technique. Row and column generation also found applications in Factored MDPs (Guestrin and Gordon 2002) and security games (Jain et al. 2010), as well as heuristic search for stochastic shortest path problems (Trevizan et al. 2016). The latter uses heuristics to guide how an LP should be expanded with variables and constraints. An important difference in our work is that we bootstrap from a previous LP, rather than expanding one individual LP.

For MOMDPs our paper focuses on finding a convex hull (CH), which is the optimal solution set for linear utility functions with unknown weights. For such problems two types of algorithms exist (Roijers and Whiteson 2017): outer loop methods and inner loop methods. Outer loop methods work by using a single-objective solver, and solving scalarized instances of MOMDPs, i.e., MDPs, to construct an (approximate) CH (Roijers, Whiteson, and Oliehoek 2015). This typically scales well in the number of states. In this paper we focussed on improving inner loop methods, which employ operators like cross-sum and pruning, to make Bellman backups work with sets of value vectors. Such methods typically scale much better in the number of objectives.

Conclusion

We proposed Bootstrap LP (BLP), a method for speeding up value iteration (VI) algorithms that require maintaining sets of value vectors, such as CHVI for MOMDPs and incremental pruning for POMDPs. Our key insight is that LP constraints that led to the final solutions in an iteration of VI can be reused to speed up the LPs in the next iteration. We have shown that BLP improves the state-of-the-art ICG algorithm (Walraven and Spaan 2017) for incremental pruning in POMDPs. Moreover, we have shown that where ICG fails to improve over simpler pruning algorithms for MOMDPs, i.e., White and Lark's (1991), BLP achieves significant speed-ups. For MOMDPs, BLP scales much better in the number of states and objectives, making BLP an important advancement of the state-of-the-art in MOMDPs. Finally, because our BLP works well for both MOMDPs and POMDPs, we believe that BLP would speed up any LP-based VI algorithm.

We aim to extend our work to reinforcement learning in MOMDPs and investigate the effects of LP reuse in model-free (Hiraoka, Yoshida, and Mishima 2009) and model-based (Wiering, Withagen, and Drugan 2014) approaches. We expect that especially in model-based approaches where the MOMDP model, i.e., the transition and reward functions, is incrementally updated with the added interactions with the environment, bootstrapping LPs from the planning step before a given model update can drastically improve performance. This would be in addition to bootstrapping the LPs from previous iterations within a given planning step. Furthermore, we aim to create methods that produce a bounded approximate solution based on bootstrapping LPs for both POMDPs and MOMDPs, building on methods like EVA (Varakantham et al. 2007).

Acknowledgments

Diederik M. Roijers is a postdoctoral fellow of the Research Foundation – Flanders (FWO). The research by Erwin Walraven is funded by the Netherlands Organisation for Scientific Research (NWO), as part of the Uncertainty Reduction in Smart Energy Systems (URSES) program.

References

Barrett, L., and Narayanan, S. 2008. Learning all optimal policies with multiple criteria. In *ICML*, 41–47.

Bellman, R. E. 1957. *Dynamic Programming*. Princeton university press.

Benders, J. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* 4(1):238–252.

Cassandra, A. R.; Littman, M. L.; and Zhang, N. L. 1997. Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. In *UAI*.

Dujardin, Y.; Dietterich, T.; and Chadès, I. 2015. α-min: A Compact Approximate Solver For Finite-Horizon POMDPs. In *IJCAI*, 2582–2588.

Feng, Z., and Zilberstein, S. 2004. Region-based incremental pruning for POMDPs. In *UAI*, 146–153.

Gilmore, P. C., and Gomory, R. E. 1961. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research* 9(6):849–859.

Guestrin, C., and Gordon, G. 2002. Distributed Planning in Hierarchical Factored MDPs. In *UAI*, 197–206.

Hiraoka, K.; Yoshida, M.; and Mishima, T. 2009. Parallel reinforcement learning for weighted multi-criteria model with adaptive margin. *Cognitive Neurodynamics* 3:17–24.

Jain, M.; Kardes, E.; Kiekintveld, C.; Ordónez, F.; and Tambe, M. 2010. Security Games with Arbitrary Schedules: A Branch and Price Approach. In *AAAI*, 792–797.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101(1):99–134.

Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In *ICML*, 157–163.

Littman, M. L. 1996. *Algorithms for Sequential Decision Making*. Ph.D. Dissertation, Brown University.

Littman, M. L. 2001. Friend-or-foe Q-learning in general-sum games. In *ICML*, 322–328.

Puterman, M. L. 1994. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. New York, NY: John Wiley & Sons, Inc.

Raphael, C., and Shani, G. 2012. The Skyline algorithm for POMDP value function pruning. *Annals of Mathematics and Artificial Intelligence* 65(1):61–77.

Roijers, D. M., and Whiteson, S. 2017. Multi-objective decision making. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 11(1):1–129.

Roijers, D. M.; Vamplew, P.; Whiteson, S.; and Dazeley, R. 2013. A survey of multi-objective sequential decision-making. *JAIR* 48:67–113.

Roijers, D. M.; Whiteson, S.; and Oliehoek, F. A. 2015. Computing convex coverage sets for faster multi-objective coordination. *JAIR* 52:399–443.

Sondik, E. J. 1971. *The optimal control of partially observable Markov processes.* Ph.D. Dissertation, Stanford University.

Trevizan, F. W.; Thiébaux, S.; Santana, P. H.; and Williams, B. C. 2016. Heuristic search in dual space for constrained stochastic shortest path problems. In *ICAPS*, 326–334.

Vamplew, P.; Dazeley, R.; Berry, A.; Dekker, E.; and Issabekov, R. 2011. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning* 84(1-2):51–80.

Vamplew, P.; Webb, D.; Zintgraf, L. M.; Roijers, D. M.; Dazeley, R.; Issabekov, R.; and Dekker, E. 2017. MORL-Glue: A benchmark suite for multi-objective reinforcement learning. In *BNAIC*, 389–390.

Varakantham, P. R.; Maheswaran, R.; Gupta, T.; and Tambe, M. 2007. Towards efficient computation of quality bounded solutions in POMDPs: Expected value approximation and dynamic disjunctive beliefs. In *IJCAI*, 2638–2643.

Walraven, E., and Spaan, M. T. J. 2017. Accelerated Vector Pruning for Optimal POMDP Solvers. In *AAAI*, 3672–3678.

White, C. C. 1991. A survey of solution techniques for the partially observed Markov decision process. *Annals of Operations Research* 32(1):215–230.

Wiering, M. A.; Withagen, M.; and Drugan, M. M. 2014. Model-based multi-objective reinforcement learning. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*.