Computing Optimal Stationary Policies for Multi-Objective Markov Decision Processes

Marco A. Wiering and Edwin D. de Jong
Intelligent Systems Group and Large Distributed Databases Group
Department of Information and Computing Sciences, Utrecht University
Padualaan 14, 3508TB Utrecht, The Netherlands
tel: +31 302539209, fax: +31 302513791, e-mail : {marco,dejong}@cs.uu.nl

Abstract— This paper describes a novel algorithm called CON-MODP for computing Pareto optimal policies for deterministic multi-objective sequential decision problems. CON-MODP is a value iteration based multi-objective dynamic programming algorithm that only computes stationary policies. We observe that for guaranteeing convergence to the unique Pareto optimal set of deterministic stationary policies, the algorithm needs to perform a policy evaluation step on particular policies that are inconsistent in a single state that is being expanded. We prove that the algorithm converges to the Pareto optimal set of value functions and policies for deterministic infinite horizon discounted multi-objective Markov decision processes. Experiments show that CON-MODP is much faster than previous multi-objective value iteration algorithms.

I. INTRODUCTION

Many real-world problems involve multiple objectives for evaluating possible (sequences of) decisions. For some sequential decision problems the multiple objectives can be combined in an a-priori defined way. However, many other environments involve multiple outcomes such as assignments of rewards to agents in a multi-agent setting, multiple emotions for an agent that determine different desires or goals, or other control systems in which multiple criteria should be optimized, and there is not always an easy or unique way to trade off the different objectives.

Although most research in optimal control [1] and reinforcement learning [6], [8] involves a single scalar reward value to define the goal for an agent, there has been some research in using multiple objectives in a reinforcement learning setting [5], [4], [12], [3], [7]. This work has usually concentrated on a fixed strategy to order the different policies such as using preference relations for ordering the different objectives and value functions [3], an a-priori known weighting function to transform the multiple criteria into a single one [5], [4], or optimizing the multiple criteria individually [12].

Research in the early 80's has already focused on multi-objective Markov decision processes [11], [2], [9] for which multi-objective dynamic programming (MODP) has been proposed. It is well known that computing optimal policies for known finite single objective Markov decision processes can be done with Dynamic Programming (DP) algorithms [1]. In Markov decision processes (MDPs) there is one scalar reward signal that is emitted after each action of an agent. Convergence proofs of DP methods applied to MDPs rely on showing contraction to a single optimal value function. In the case of multi-objective MDPs there is not a single optimal policy, but a set of Pareto optimal policies that are not dominated by any other policy. We will formally define a dominance operator in Section 3, but now it suffices to say that a policy is dominated by another policy if that other policy achieves more long term reward on at least one objective and no less long term reward on other objectives. White (1982) proved convergence for a value iteration algorithm working on Pareto optimal sets of policies for infinite horizon problems. However, his algorithm computes non-stationary policies and is computationally infeasible for computing all Pareto optimal policies for an infinite horizon problem with a large discount factor.

In this paper we propose a new algorithm for efficiently computing all deterministic stationary Pareto optimal policies. We will use the terms stationary and consistent interchangeably in this paper, meaning that in a particular state always the same action is selected. The novel algorithm CON-MODP is based on a consistency operator that only allows stationary policies to remain in the Pareto optimal set and a policy evaluation step to deal with policies that are only inconsistent in a single state that is being expanded. In this paper, we assume deterministic environments for describing the algorithm and doing experiments.

Outline. Section II describes dynamic programming and Section III describes multi-objective dynamic programming. Then Section IV describes the CON-MODP algorithm for computing stationary Pareto optimal policy sets and value functions. In Section V experimental results are shown on a number of toy problems. Section VI concludes this paper.

II. DYNAMIC PROGRAMMING

Dynamic Programming algorithms are able to compute an optimal policy for an agent given the specification of a Markov decision process (MDP). A finite MDP is defined as; (1) The state-space $S = \{s^1, s^2, \ldots, s^n\}$, where $s_t \in S$ denotes the state of the environment at time t; (2) A set of actions available to the agent in each state $A(s)$, where $a_t \in A(s_t)$ denotes the action executed by the agent at time t; (3) A transition function $P(s, a, s')$ mapping state-action pairs s, a to a probability distribution over successor states s'; (4) A reward function $R(s, a, s')$ that denotes the expected reward obtained when the

agent makes a transition from state s to state s' using action a, where r_t denotes the (possibly stochastic) scalar reward obtained at time t; (5) A discount factor $0 \leq \gamma < 1$ that discounts later rewards compared to immediate rewards.

In optimal control we are interested in computing the optimal policy for mapping states to actions. We denote the optimal deterministic policy as $\pi^*(s) \rightarrow a^*|s$. It is well known that for each MDP, one or more optimal deterministic policies exist. The optimal policy is defined as the policy that receives the highest cumulative discounted rewards in its future from all states. In order to compute the optimal policy, dynamic programming algorithms [1], [8] use value functions to compute the expected utility of selecting particular actions given a state. We denote the value of a state $V^\pi(s)$ as the expected cumulative discounted future reward when the agent starts in state s and follows a policy π:

$$V^\pi(s) = E(\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, \pi)$$

In most cases dynamic programming algorithms used for computing a control policy also make use of a Q-function for evaluating state-action pairs. Here $Q^\pi(s, a)$ is defined as the expected cumulative discounted future reward if the agent is in state s, executes action a, and follows policy π afterwards:

$$Q^\pi(s, a) = E(\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, a_0 = a, \pi)$$

If the optimal Q-function Q^* is known, the agent can select optimal actions by selecting the action with the largest value in a state: $\pi^*(s) = \arg\max_a Q^*(s, a)$. Furthermore the optimal value of a state should correspond to the highest action value in that state according to the optimal Q-function: $V^*(s) = \max_a Q^*(s, a)$.

It is well known that there exists a recursive equation known as the Bellman optimality equation [1] that relates a state-action value of the optimal value function to other optimal state values that can be reached using a single transition:

$$Q^*(s, a) = \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

The Bellman equation has led to very efficient dynamic programming (DP) techniques for solving known MDPs [1], [8]. One of the most used DP algorithms is value iteration which uses the Bellman equation as an update:

$$Q^{k+1}(s, a) := \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V^k(s'))$$

Where $V^k(s) = \max_a Q^k(s, a)$. In each step the Q-function looks ahead one step using this recursive update rule. It can be shown that $\lim_{k \to \infty} Q^k = Q^*$ when starting from an arbitrary bounded Q^0.

III. Multi-objective Dynamic Programming

In multi-objective Markov decision processes (MOMDPs), instead of the usual scalar reward signal r, a reward vector $\vec{r}$ is used where the length of $\vec{r}$ denoted as l is the number of dimensions or components of the reward function. For each transition from a state-action pair to a next state, a reward function $\vec{R}(s, a, s') = (R_1(s, a, s'), \ldots R_l(s, a, s'))$ is given.

Since the reward vector in MOMDPs has multiple components, there naturally arise conflicts between them. For example, it may be possible that an agent is able to compute a policy for buying the cheapest possible flight ticket, but that the duration of this flight is longer than other possible flights. Therefore, we have to deal with several trade-offs to finally select a policy (or plan). The solution is to compute all policies that are not dominated by another policy. This set of policies is called the *Pareto efficient (optimal) set*. When given the set of Pareto optimal policies, we can let a user or agent select one of them.

To combine the different reward components, we may also introduce a linear weighting function that determines a collective reward $R(s, a, s') = \sum_{i=1}^{l} w_i R_i(s, a, s')$ using a weight vector $\vec{w}$. In the following we assume weight vectors with positive elements only which are constant over states. If $\vec{w}$ would be known beforehand and would always remain the same, the problem would reduce to a normal MDP after computing the collective reward function. However, we suppose that $\vec{w}$ is unknown and can be selected by a user or agent *after* the MOMDP is solved by the algorithm. Once $\vec{w}$ is known, we require the algorithm to efficiently compute an optimal policy. This means that if $\vec{w}$ would change during the interaction of the agent with the environment, the new $\vec{w}$ can be immediately used without the necessity of a lot of computation.

To solve this problem we can use particular multi-objective dynamic programming (MODP) algorithms that compute the Pareto front of optimal policies and corresponding value functions. These MODP methods are based on value iteration [11] or policy iteration [9], [2]. The problem when one would like to use conventional dynamic programming is that there are multiple Pareto optimal policies, and therefore the algorithm has to work with sets of policies and value functions. In this section we will discuss a possible naive approach to solving this problem, and then we will describe White's algorithm (1982) and its main disadvantage. White's algorithm uses value iteration and therefore is closer in spirit to our method than other policy iteration methods.

Problem with solving separate MDPs. One naive idea to cope with the problem of dealing with multiple objectives is to make multiple MDPs, each with only one single component of the reward vector, solve these separate MDPs using DP and then use the weight vector once it is available to select actions. Although this solution is simple, it will not work for all MOMDPs. We will explain this by an example, see Fig. 1.

When we use dynamic programming on the different components, then both policies will choose to go to state A when the agent is in state B. However, state C promises a higher combined reward for many weighting functions (e.g., when $\max_i w_i < 0.7$ and $\sum_i w_i = 1$). Thus, the policy will never go to state C from B using this method, and the agent gets less cumulative reward for many weighting functions.
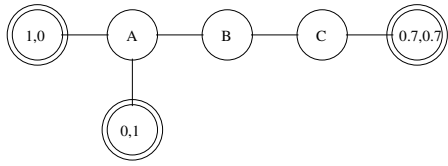
Fig. 1. *This example shows why computing separate policies for different reward components does not work. In the terminal states the numbers show emitted rewards for the 2 reward vector components.*

White's multi-objective DP algorithm. The solution of most multi-objective DP methods is to keep track of the non-dominated (or Pareto efficient) set of value functions. The value function now has different cumulative reward components or values and this is denoted by a value function $V^i(s) = (V_1^i(s), V_2^i(s), \ldots, V_l^i(s))$, where V_x^i denotes the discounted cumulative reward intake of reward component x of policy i. Furthermore, we use a set V^O for the set of value functions (and policies) that are not dominated:

$$V^O(s) = \{V^i(s) | V^i(s) \text{ isn't dominated by a policy in } s\}$$

Below we will formally define the non-dominated operator. We will also use V^D for denoting the set of value functions computed at some given moment that may include dominated ones. The same is done for the Q-function, thus we keep track of a set Q^O that denotes the set of non-dominated Q-functions. Here the Q-functions should not be dominated by another Q-vector for the same state-action pair. Thus:

$$Q^O(s, a) = \{Q^i(s, a) | Q^i(s, a) \text{ isn't dominated in } s, a\}$$

We will now formally define a dominance function $\succ$ that works on two value vectors for a state s as follows:

$$V^i(s) \succ V^j(s) \Leftrightarrow \exists x; V_x^i(s) > V_x^j(s) \wedge \neg \exists y; V_y^i(s) < V_y^j(s)$$

So a value vector of policy i dominates a value vector of policy j if i has a higher value on some component x and does not have a lower value on any other component.

In our implementation we discovered that it could happen that two different actions had the same result (transition to the next state and same reward). This sometimes led to an explosion of the number of policies with the same reward intakes on all components. To take care of this, we used the following dominance function $\succ'$ in our experiments:

$$V^i(s) \succ' V^j(s) \Leftrightarrow \neg \exists y; V_y^i(s) < V_y^j(s) \wedge i < j$$

Where the policies are ordered (using the indices i and j). So here a policy is dominated by another one if it does not receive more long term reward on some component and its index is larger. This second definition significantly increased the speed of the algorithm without loosing optimality (the long term reward intake of some of these policies is exactly the same, so we can keep only one of them).

Now we define a non-dominated operator ND that tells whether a policy i is not dominated in the state s by any value function vector of the set of value functions $V^D(s)$:

$$ND(V^i(s), V^D(s)) \Leftrightarrow \neg \exists V^j(s) \in V^D(s) ; V^j(s) \succ V^i(s)$$

Analogous definitions hold for Q-vector functions. Now it becomes possible to enhance dynamic programming to compute non-dominated value function sets. For simplicity we restrict ourselves to deterministic MDPs. We define the Pareto optimal operator PO as:

$$\begin{aligned} PO(Q^D(s,a)) &= \{Q^i(s,a) | Q^i(s,a) \in Q^D(s,a) \wedge \\ & \quad ND(Q^i(s,a), Q^D(s,a))\} \end{aligned}$$

We also define the operator PO to work on all state-action pairs. Furthermore, we construct the dynamic programming operator as follows where $\oplus$ denotes an addition operator working on sets (and vectors) and s' is the next state:

$$DP(Q^D(s,a)) = (\vec{R}(s,a,s') \oplus \gamma V^O(s') | P(s,a,s') = 1.0)$$

Here $V^O(s)$ is computed as:

$$V^O(s) = PO(\cup_a Q^D(s,a))$$

Note that the usual $\max$ operator in dynamic programming is replaced by the Pareto optimal operator working on the union of Q-value vectors. Since the PO operator will not always keep a single value vector, the policy set will usually grow after multiple iterations until it finally converges. We summarize this algorithm to compute the optimal set of Q-value vectors Q^{O*} as:

$$Q^{O*} = (PO(DP(Q_0)))^*$$

Where $(X)^*$ denotes that operator X is repeated an infinite number of times, but which in reality is repeated a finite number of times for a discount factor smaller than one (this can be seen by the fact that actions selected far away in the future will be discounted so much that their contribution becomes smaller than the fixed machine precision). We also define $(X(Y))^0 = Y$.

White (1982) already proved that iterating the algorithm using the PO operator that eliminates all dominated policies converges and leads to the unique set of Pareto optimal policies. Here, we will give another novel proof that shows that the set of final Pareto optimal policies computed by the algorithm is the same as when the set of Pareto optimal policies is computed after computing the value functions for all possible policies:

$$PO(DP'(Q_0))^* = (PO(DP(Q_0)))^* \quad (1)$$

We need to define another dynamic programming operator DP' that does not make use of the PO operator at all:

$$DP'(Q^D(s,a)) = (\vec{R}(s,a,s') \oplus \gamma V^D(s') | P(s,a,s') = 1.0)$$

Where $V^D(s)$ is computed as:

$$V^D(s) = \cup_a Q^D(s,a)$$

Note the difference between the two algorithms shown in Equation (1): the one at the left side only uses the Pareto optimal operator one time after all infinite horizon Q-value vectors have been computed, and White's algorithm (on the right) uses the Pareto optimal operator after every step (thereby

significantly reducing computational cost since during computation less policies are expanded).

First we must assure that there are a finite number of solutions that can be computed using both algorithms. This is clear from the fact that the discount factor is smaller than 1. Therefore, even for the algorithm $(DP'(Q_0))^*$ (that computes most policies) we only need to consider n-step horizons, since the actions computed for a n+1-step horizon has its last action discounted using γ^{n+1}. If this value becomes smaller than the finite machine precision ϵ then it will not contribute anything to the value of the policy. Therefore, it suffices to prove the following theorem:

Theorem 1.

$$PO(DP'(Q_0))^n = (PO(DP(Q_0)))^n$$

Where $n = \lceil \frac{\log \epsilon}{\log \gamma} \rceil$. For our proof we make use of the following Lemma:

Lemma 1. Non-optimal temporally local sub-policies do not need to be kept in order to compute an optimal global policy.

This lemma requires some more explanation. With a global policy we mean a policy that computes actions for an infinite horizon (here approximated by n). With a temporally local sub-policy we mean a policy that only computes actions for a finite (e.g. m-step) horizon. We do not need to keep the results of all m-step policies to compute the optimal global policy, this is exactly what makes dynamic programming efficient. The reason is that when the agent arrives in a state and selects an m-step policy, it will always choose the best one. If we look at Fig. 2 we can see that the optimal global policy goes from state A to B and from state B to state A. It is true that for a 1-step horizon going from A to B is dominated by the action that takes the agent to state A from state A. After two dynamic programming iterations for computing the 2-step horizon policies, however, we will expand state A again and the algorithm will find out that it is better to go to state B from state A. Thus, local sub-policies that are dominated do not need to be kept.
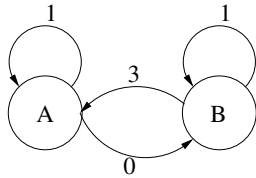

Fig. 2. *Example used to explain the workings of dynamic programming. In Section 4 this example is used to show why naively computing consistent non-dominated policies does not work.*

Proof 1. We will show that an element of $L_1^n = PO(DP'(Q_0))^n$ is a member of $L_2^n = (PO(DP(Q_0)))^n$ and vice versa. First we prove $L_1^n \subseteq L_2^n$. Suppose that there is a member $x \in L_1^n$ and suppose that there is no equivalent member $x' \in L_2^n$. This can be the case due to: (1) There is a $y \in (DP'(Q_0))^{n-1}$ from which x was constructed that does not have a corresponding member $y' \in L_2^{n-1} = (PO(DP(Q_0)))^{n-1}$ or (2) x' is dominated by some solution

in L_2^n and x is not dominated in L_1^n. We first consider (2). Since the dominance comes from the solutions before the PO operator is used, we can easily see that dominance in L_1^n comes from a solution that is inside $(DP'(Q_0))^n$. Since this contains all n-step policies, it is much bigger and therefore there is always as much or more dominance in L_1, and thus (2) cannot be the case. Now let's consider (1): we note that it can be the case that there is a solution $y \in (DP(Q_0))^{n-1}$ that does not have an equivalent member $y' \in L_2^{n-1}$. However, if y' is not a member of $PO(DP'(Q_0))^{n-1}$ it does not need to be kept according to lemma (1) to compute solutions for L_1^n. Therefore we only have to consider solutions $y' \in PO(DP'(Q_0))^{n-1}$. We note that such solution y' must exist since otherwise we can observe that if $PO(DP'(Q_0))^n \not\subseteq L_2^n$ then also $PO(DP'(Q_0))^{n-1} \not\subseteq L_2^{n-1}$. But this leads to a contradiction since $L_1^0 = L_2^0 = Q_0$.

Now we prove $L_2^n \subseteq L_1^n$. Suppose that there is a member $x \in L_2^n$ and suppose that there is no equivalent member $x' \in L_2^n$. This can be the case due to: (1) There is a $y \in L_1^{n-1}$ from which x was constructed that does not have a corresponding member $y' \in (DP(Q_0))^{n-1}$, or (2) x' is dominated by some solution in L_1^n and x is not dominated in L_2^n. We first consider (1) and note that $(DP(Q_0))^{n-1}$ contains all n-1 step policies, therefore y' will always be a member of it if y exists. Now we consider (2). We first note that the dominance function is transitive and therefore we only have to examine dominance by non-dominated solutions. Since we already proved before that $L_1^n \subseteq L_2^n$, we can see that there is no less dominance in L_2^n and therefore if x' would be dominated then so would x. This concludes our proof.

Although the algorithm will finally converge to all optimal policies, the problem of this algorithm is that policies are allowed to be non-stationary. Since the number of non-stationary policies increases exponentially with the lookahead horizon, the Pareto optimal policy set will increase very fast and thus this algorithm can only be used on very simple problems with low values for the discount factor. For example, if we use this algorithm on the problem shown in Fig. 3, we will get all sequences $A, A, A, \ldots, A, A, B, \ldots$, etc. as outcome in the computed policies and value functions since none of them is dominated by another one (the resulting number of Pareto optimal solutions depends on the discount factor that determines the lookahead horizon).
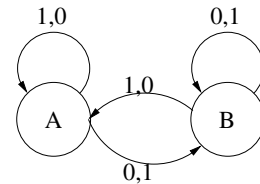

Fig. 3. *An example that shows why computing all Pareto optimal non-stationary policies is computationally infeasible in general.*

IV. CON-MODP

In infinite horizon discounted Markov decision processes, there is always a single optimal value function and one or more stationary deterministic policies belonging to it. Therefore, if finally a specific set of weights is selected by a user or agent after the algorithm has computed all Pareto optimal policies, we know that there is a stationary and deterministic policy that is optimal. Our novel algorithm CON-MODP focuses on only computing stationary deterministic Pareto optimal policies. For dealing with non-stationary policies, the algorithm uses a consistency operator that eliminates most non-stationary policies and reevaluates non-stationary (or inconsistent) policies that are only inconsistent in 1 single state that is being expanded.

Instead of only keeping track of a Q-vector $Q^i(s, a)$ the algorithm also keeps track of the policy π_i so that we can detect whether a policy is consistent (stationary) or not. These policies are stored with state-action pairs. A policy becomes inconsistent if $Q^i(s, a)$ is computed and if the policy π_j of the value vector of the next state $V^j(s')$ does not select action a in state s. This can happen easily since in the beginning for a limited horizon it may look promising to select action b in state s by a local optimal sub-policy and later the algorithm tries to make a lookahead evaluation step by selecting action a in state s and connect this action with the previous policy. One possible solution is to expand the algorithm by changing the definition of non-dominated: a policy is non-dominated if it is consistent and non-dominated in the previous sense. If we would do this, however, the algorithm would not be able to compute all Pareto optimal policies.

This problem is explained using Fig. 2. For simplicity the example only uses a single scalar reward for which the problem also occurs, and the discount factor is set to 1. Table I shows the computational steps performed by the naive algorithm on the problem shown in Fig. 2. The table shows the planning horizon t and the Q-values for the different transitions (e.g. from state B to state A). N.C. stands for not-consistent and therefore these policies are eliminated. For example in time-step 2 $Q(B, B)$ first goes from state B to state B and then tries to connect it to the value function $V(B)$ of state B computed for a 1-step horizon that chooses to go to state A. This leads to an inconsistency. After some time all policies become inconsistent and are eliminated.

TABLE I

Results for the naive algorithm that does not reevaluate inconsistent policies, but only keeps consistent policies. For the problem shown in Fig. 2, this leads to a collapse of the solution space.

t	1	2	3
Q(A,A)	1	2	N.C.
Q(A,B)	0	3	N.C.
Q(B,B)	1	N.C.	N.C.
Q(B,A)	3	4	6
V(A)	1 ($A \to A$)	3 ($A \to B, B \to A$)	N.C.
V(B)	3 ($B \to A$)	4 ($B \to A, A \to A$)	6 ($B \to A, A \to B$)

Policies become inconsistent due to the last chosen action of the Q-value vector that is expanded. If we eliminate them

all by our (wrong) definition of non-dominated, the algorithm is not able to compute the optimal policies for all problems. Note that conventional value iteration also computes value functions of inconsistent policies as long as the algorithm is not finished. However, since these value functions are not eliminated, there is no problem since after many lookahead steps this inconsistency will disappear. The results of initially chosen inconsistent actions will be discounted so much that they will not have any influence on the final value functions.

The solution is to compute consistent non-dominated policies with reevaluating inconsistent policies. CON-MODP detects when a policy is made inconsistent due to the last lookahead update step, and then changes it to a consistent policy by forcing the action in the current state that is evaluated all the times this state will be visited. In this way, the policy is consistent again. Note that this method is the only realistic method for making the policy consistent, since forcing different actions will not always make the transition to the same next state, and if they did, they would also be evaluated by the DP operator. Since there is no direct way of knowing the value vector of the changed policy (it was based on the assumption that another future action in the same state would be chosen), CON-MODP uses policy evaluation to reevaluate the policy. The whole algorithm uses the operator CON next to PO and DP. We define CON that works on the Q-value vector of a state-action pair (s, a) with policy π as:

$$
\begin{aligned}
CON(Q^\pi(s, a)) \quad &= \quad \pi \text{ with } Q^\pi(s, a), \text{ if } \pi(s) = a \\
&= \quad \pi' \text{ with } Eval(\pi'), \text{ if } \pi(s) \neq a \text{ is the only} \\
&\quad\quad \text{inconsistency, and } ND(Q^\pi(s, a), Q^O(s, a)) \\
&\quad\quad \text{where } \pi'(s) = a \text{ and } \pi'(s') = \pi(s') \; \forall s' \neq s \\
&= \quad \emptyset, \text{ otherwise}
\end{aligned}
$$

Here $Eval(\pi')$ means that policy π' is evaluated using policy evaluation for the same number of steps as the original policy π is computed. Evaluation is only done if the original inconsistent policy was not already dominated. CON makes policies generated after a lookahead step consistent again. It does this by forcing the latest evaluated state's action throughout the policy. If one would not do this, the resulting policy would be non-stationary and therefore this inconsistent policy would continue to play a role in the future of the system.

We also let CON work immediately on sets of Q-functions and policies. We now define the CON-MODP algorithm as:

$$
Q^{O*} = (PO(CON(DP(Q_0))))^*
$$

This algorithm is able to compute all stationary Pareto optimal policies and corresponding value functions for deterministic finite MOMDPs. To show this, we prove that:

$$
(PO(CON(DP(Q_0))))^* = PO(CON(DP(Q_0))^*)
$$

The proof is very similar to proof 1 by noting that instead of only the PO operator we now use the PO and CON operators. Both will eliminate policies. The lemma which we use now is that we do not need to keep temporally local sub-policies that are dominated or that are non-stationary. The additional non-stationary requirement in the lemma can be easily shown to be valid, since if we keep such non-stationary policies they need to be eliminated at some time in order

to compute the stationary non-dominated policies. We will not repeat proof 1 here, the extension we need is to use the CON operator before the PO operator and everything else remains equal. Thus, our algorithm computes optimal value functions and policies, but we cannot yet say anything about the convergence speed which will also depend on the discount factor and the number of reward components.

The solution strategy of CON-MODP is demonstrated in Table II on the example shown in Fig. 2. Reevaluation is done by computing the value function for the same lookahead as the number of steps the dynamic programming algorithm has iterated. This is to ensure that the policies are comparable for finite horizons. To decrease the number of reevaluations, we only reevaluate an inconsistent policy if the computed value function based on an inconsistent policy was not already dominated by the value function of a consistent policy (in Table II reevaluations eval* are not really performed). For example in time-step 2, we expand $Q(B,B)$ so that first a transition to B is made, then it is discovered that connecting this action to the policy of state B is inconsistent. Therefore the CON operator forces to select the action go to B two times. This last policy is reevaluated and has a value 2.

TABLE II

CON-MODP reevaluates policies that are made inconsistent due to a lookahead evaluation step. This is done for the policies that only have a single inconsistency in the last expanded state-action pair.

t	1	2	3
Q(A,A)	1	2	3 eval
Q(A,B)	0	3	3 eval
Q(B,B)	1	2 eval	3 eval*
Q(B,A)	3	4	6
V(A)	1 (A → A)	3 (A → B,B → A)	3 (B → A, A → B)
			3 (A → A)
V(B)	3 (B → A)	4 (B → A, A → A)	6 (B → A, A → B)

The main drawback is that policy evaluation is quite costly, although it is still quite fast for deterministic MDPs. To improve the efficiency, different datastructures such as policy-trees or hash-tables can be used to keep previous results of evaluated policies.

Selecting optimal actions using the Pareto set. Once the algorithm has computed the Pareto set and the weight vector $\vec{w}$ is known, it is easy to select optimal actions. For this the algorithm walks through the set of Pareto optimal Q-values for each state s and all actions a and then computes the maximal combined Q-value using the known weight $\vec{w}$. Then the best action in each state is stored in the current optimal policy for the weight vector.

Extension to stochastic MDPs. In case there are stochastic transitions, the problem becomes much harder. The reason is that given some state-action pair, there are transitions to multiple next states and then the algorithm has to merge elements of the non-dominated sets of these next states, which is expensive if there are many possible next states with many non-dominated value functions. Another difficulty is that it becomes harder to detect inconsistent policies. It is possible that the value vectors of next states need to be combined that

have inconsistent policies. The algorithm should make these policies consistent in order to evaluate the result, however there can be many inconsistencies and many ways to make them consistent. If one simply discards the resulting inconsistent policies, it may be impossible to compute all Pareto optimal solutions. It is also possible to make the connection with partially observable Markov decision processes [10]. However, in partially observable MDPs there are other kind of pruning possibilities, e.g. if some policy is dominated by a combination of two other policies, then the dominated policy can be pruned. In MOMDPs in general we are looking for the compelte set of non-dominated policies. Although in this article we assume linear weighting vectors, in general the weighting rule does not need to be a linear function. That means that even if the Pareto front is convex, we cannot prune the policies lying in between the extremes. Therefore, further pruning of our non-dominated set of policies is impossible in the current framework.

V. EXPERIMENTS

To verify the CON-MODP algorithm, we made randomly connected deterministic MOMDPs, and experimented with the number of states, actions, and reward components. We performed thousands of simulations to compare the final obtained Pareto front to solving an MOMDP when the weighting function is known, and as expected, the CON-MODP algorithm always computed the optimal policy. Furthermore, the algorithm always converged to a stable Pareto optimal set. So as results, we observe the resulting size and combined values of the Pareto front of the value functions V^O which indicates how many policies have to be stored after convergence, and we examine the time needed for the algorithm to converge (actually we stop iterating if the previous sum of Pareto optimal value functions of all states does not change more than 10^{-5}).

Experimental set-up. We use $|S|$ states and $|A|$ actions possible in each state. For each state-action pair we generate a random successor state from S. After this, we check whether the structure of the MDP is ergodic by running a random policy and checking whether it visits all states more than 1 time. On each transition we generate a random reward vector where each reward component is 0 with probability 0.75 and otherwise it has a random value between 0 and 1.

Experimental results. We first show experiments on a simple randomly generated problem in which we compare White's method that computes non-stationary Pareto optimal policies to our method that computes stationary Pareto optimal policies. We also use an algorithm that follows White's algorithm for 18 steps and then switches to using CON-MODP. The experiments were done for a simple randomly generated problem consisting of 5 states, 2 reward components, 3 actions, and a discount factor of 0.25. Fig. 4 shows the results of displaying the sum of state-values for all Pareto optimal policies, states, and reward components. We can see that all methods converge, but that our method keeps the number of stored Pareto optimal policies much lower and therefore the sum of state-values as well. The mixed algorithm (consistent 18 steps) also converges
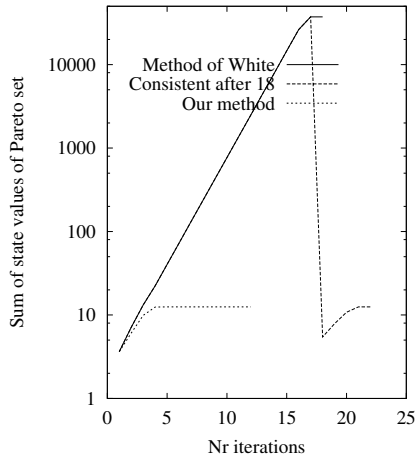
Fig. 4. *The sum of state-values of all Pareto optimal policies for the different algorithms for a problem with 5 states, 3 actions, 2 reward components, and $\gamma = 0.25$. Note that White's method leads to a higher sum since much more policies are kept.*

Fig. 5. *The time needed for the different algorithms given different numbers of states with 4 actions, 3 reward components, and $\gamma = 0.95$.*

to the same policies as our algorithm, but not in a single step after 18 iterations with White's algorithm. This can be explained by the fact that we use the $\succ'$ dominance operator that may have eliminated some consistent policies since some inconsistent policies had exactly the same value. Therefore it requires some iterations to get these policies back. On the other hand this also means that CON-MODP can start with many different initial Q-value vectors and policies.

The CON-MODP algorithm converged to 21 Pareto optimal policies (note that some policies are stored multiple times, since states duplicate some stored policies). White's method, however converged to 62258 non-stationary Pareto optimal solutions. Due to this difference, the CON-MODP algorithm only needed 0.01 seconds to converge and White's method 590 seconds.

We also performed experiments to examine the computational complexity of the CON-MODP algorithm if we change the number of states, actions, and reward vector components. For this we first run experiments for environments with 3 reward vector components and 4 actions, and $\gamma = 0.95$, where we made the number of states a variable. Clearly White's method cannot be executed on these problems due to too much computational cost. The experiments were repeated 20 times for randomly generated environments with the specific number of states. We compare construction time of all Pareto optimal solutions of our method to using the naive algorithm that does not reevaluate inconsistent policies, but always eliminates them. Furthermore, we compare the average total time needed to select actions with our method after the construction of Pareto optimal solutions is done to running DP for 10,000 different weight vectors in each environment.

Fig. 5 shows that our method is faster in evaluation and action selection time than computing optimal policies from scratch each time a new weight vector is given. Our method needs a lot of construction time, however. All methods seem
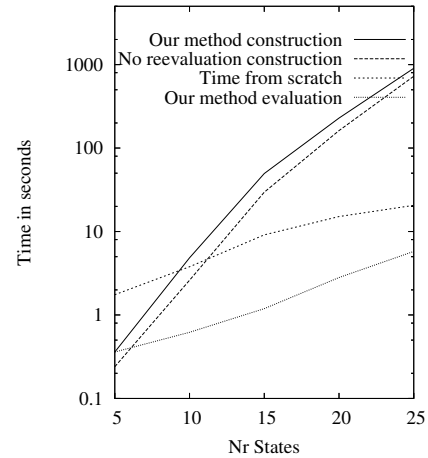
to need computation time exponential in the amount of states. This can be explained by noting that there are $|A|^{|S|}$ possible policies. The algorithm that does not reevaluate policies makes errors (was not able to compute the optimal policy) for the environments with 5, 10, 15, 20, and 25 states in 13, 15, 12, 10, and 9 percent of the tests in the 20 simulations with the 10,000 different weight vectors.

We now show experimental results while varying the number of reward components. For these experiments we used 10 states, 4 actions, and a discount factor $\gamma = 0.95$. The experiments were repeated 20 times. The results are shown in Fig. 6.
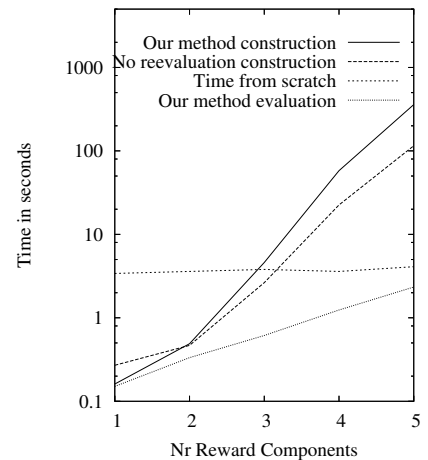

Fig. 6. *The time needed for the different algorithms given different numbers of reward components with 10 states, 4 actions, and $\gamma = 0.95$.*

We observe that CON-MODP is again faster in evaluation and action selection time than computing optimal policies from scratch each time a new weight vector is given. However, when varying the number of reward components, the computational time for computing policies from scratch is not

influenced by the number of reward components, since we only make use of the collective reward function. Therefore, if we would use a very large number of reward function components, computing policies from scratch may be finally faster in action selection time. We note that CON-MODP's computational time increases exponentially when adding more reward vector components. This is due to the growing number of policies that are not dominated when there are more reward components. The algorithm that does not reevaluate policies makes errors (was not able to compute the optimal policy) for the environments with 1, 2, 3, 4, and 5 reward components in 56, 34, 15, 11, and 5 percent of the tests in the 20 simulations with the 10,000 different weight vectors.

We finally show experimental results while varying the number of actions. For these experiments we used 10 states, 3 reward components, and a discount factor $\gamma = 0.95$. The experiments were again repeated 20 times. The results are shown in Fig. 7.
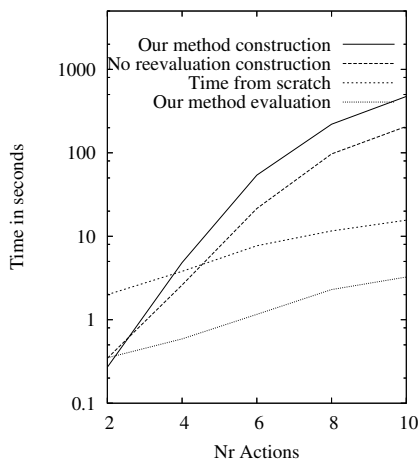

Fig. 7. *The time needed for the different algorithms given different numbers of actions with 10 states, 3 reward components, and $\gamma = 0.95$.*

The results show that CON-MODP is again faster in evaluation and action selection time than computing optimal policies from scratch. The computational time scales polynomially with the number of actions which was expected (there are $|A|^{|S|}$ policies). The algorithm that does not reevaluate policies makes errors for the environments with 2, 4, 6, 8, and 10 actions in 14, 15, 5, 8, and 6 percent of the tests in the 20 simulations with the 10,000 different weight vectors.

VI. CONCLUSIONS

We studied multi-objective Markov decision processes where a multi dimensional reward vector replaces the usual scalar reward signal. This multi-objective Markov decision process (MOMDP) can be mapped to an MDP when the weighting function of the different reward components would be known beforehand. However, in this paper we assume that the weighting function can be arbitrary and can be provided by the agent or user after the system solved the problem. To deal with this problem, we keep track of Pareto optimal stationary (consistent) policies. Our algorithm relies on computing and storing only non-dominated policies that are consistent and for this aim sometimes reevaluates policies.

Although our method works well for deterministic MOMDPs, more work has to be done to solve stochastic MOMDPs. For this, the connection with partially observable Markov decision processes can be made [10]. We are also interested in using reinforcement learning algorithms to compute Pareto optimal policy sets instead of relying on the use of DP and known MOMDPs. A good possibility would be to use model-based reinforcement learning in which first a model of the environment is learned and then an MODP algorithm can be used. We also want to focus on asynchronous dynamic programming algorithms where different states have different amounts of evaluated lookahead steps. Now it becomes harder to compare policies, since they should be evaluated over the same horizon. Finally, for huge or continuous state-action spaces, where we have to sacrifice optimality anyway, we would like to use reinforcement learning and function approximators for learning a subset of the Pareto optimal value functions. For this we want to use confidence intervals of the value functions in the dominance function.

REFERENCES

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
[2] N. Furukawa. Vector valued Markovian decision processes within countable state space. In R. Hartley, L.C. Thomas, and D.J. White, editors, *Recent Developments in Markov Decision Processes*, pages 205–223. Academic Press, New York, 1980.
[3] Z. Gabor, Z. Kalmar, and C. Szepesvari. Multi-criteria reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, pages 197–205. Morgan Kaufmann, 1998.
[4] S.C. Gadanho. Learning behavior-selection by emotions and cognition in a multi-goal robot task. *Journal of Machine Learning Research*, 4:385–412, 2003.
[5] M. Humphrys. Action selection methods using reinforcement learning. In Pattie Maes, Maja Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 135–144. MIT Press, Bradford Books, 1996.
[6] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
[7] S. Mannor and N. Shimkin. A geometric approach to multi-criterion reinforcement learning. *Journal of Machine Learning Research*, 5:325–360, 2004.
[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT press, Cambridge MA, A Bradford Book, 1998.
[9] L.C. Thomas. Constrained Markov decision processes as multi-objective problems. In S. French, L.C. Thomas, R. Hartley, and D.J. White, editors, *Multi-Objective Decision Making*, pages 77–94. Academic Press, 1983.
[10] C.C. White and W.K. Kwang. Solution procedures for vector criterion Markov decision processes. *Large Scale Systems*, 1:129–140, 1980.
[11] D.J. White. Multi-objective infinite-horizon discounted Markov decision processes. *Journal of Mathematical Analysis and Applications*, 89:639–647, 1982.
[12] W. Zhou and R. Coggins. Multiple sources of reward hierarchical reinforcement learning. In *Proceedings of the International Conference on Computational Intelligence for Modelling*, pages 934–945, 2004.