A Temporal Difference Method for Multi-Objective Reinforcement Learning

Manuela Ruiz-Montiel[1], Lawrence Mandow, José-Luis Pérez-de-la-Cruz

Andalucía Tech, Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, España.

Abstract

This work describes MPQ-learning, an temporal-difference method that approximates the set of all non-dominated policies in multi-objective Markov decision problems, where rewards are vectors and each component stands for an objective to maximize. Unlike other approximations to Multi-objective Reinforcement Learning, MPQ-learning does not require additional parameters or preference information, and can be applied to non-convex Pareto frontiers. We also present the results of the application of MPQ-learning to some benchmark problems and compare it to a linearization procedure.

Keywords: Reinforcement Learning, Multi-Objective Optimization, MOMDPs, Q-learning

1. Introduction

Markov decision processes (MDPs) are sequential decision problems appearing in many real-world situations. Solving an MDP involves defining a policy, i.e. deciding which action to take in every situation. Each action has a probabilistc outcome and may result in a certain reward. Therefore, an optimal solution is to find a policy that maximizes expected accumulated reward. Most work in this area is aimed to solve MDPs with scalar rewards. However, many problems are best formulated as multicriteria MDPs where

*Corresponding author.

Email addresses: mruiz@lcc.uma.es (Manuela Ruiz-Montiel), lawrence@lcc.uma.es (Lawrence Mandow), perez@lcc.uma.es (José-Luis Pérez-de-la-Cruz)

rewards are vectors, and components stand for different, possibly conflicting scalar objectives. These are usually referred to as Multiobjective Markov Decision Processes (MOMDPs). Dynamic programming can be used under certain assumptions to optimally solve MOMDPs when a complete probabilistic model of actions is available[16] [17] [2]. Alternatively, reinforcement learning techniques can be applied when such knowledge is not available, so yielding the field of multiobjective reinforcement learning (MORL) [13] [10].

In MORL it is now usual to distinguish between single policy and multiple policy approaches [13]. In the former we are interested in learning just *one* policy —the one that best satisfies a certain preference structure. On the contrary, in the multiple policy approach we are interested in learning (an approximation of) the Pareto front, that is, in learning *a set* of different policies. The algorithm presented in this paper is designed to learn all the Pareto front, so it can be seen as following the multiple policy approach. It should be noticed that even in the case of the single-policy approach it can be conceptually necessary or computationally useful to approximate the Pareto front as a first step [10], as the scalarization function can be non-linear or preferences can be hard to define a priori

Many proposals for approximating the Pareto front in fact approximate the subset given by its supported points. Supported points are solutions to a single objective MDP where the objective is a linear combination of the original ones. This approach has been adopted when a complete model is available [17] [2] and also when it is necessary to apply RL techniques [3], [9]. Since for single objective infinite horizon MDPs there always exists an optimal policy that is deterministic and stationary [5], these proposals just find deterministic and stationary policies. That result does not hold for MOMDPs [16] and, in general, non deterministic and/or non stationary policies can yield new points in the Pareto front. However, ([10], Corollary 1) these points will never lie outside the convex hull determined by deterministic stationary policies.

If we allow mixtures of deterministic policies, then we can consider just the convex hull [12]. But there are situations where such mixtures would not be acceptable for ethical reasons [6] or would be unfeasible, as in the case of problems where we are not only interested in the expected value of the learned policies, but also in their particular sequence of states and actions. Allowing stochastic policies in these cases can result in missing some interesting deterministic, not supported solutions. It can be the case of domains like computational design [11]. For these considerations, the

algorithm here presented learns just deterministic policies. But it learns *all* Pareto optimal deterministic policies, supported or not supported.

A final consideration should be made in order to characterize our contribution. Many proposals for multiple policy MORL follow a sequential scheme and compute only a policy at once. On the contrary, few algorithms aim to learn the set of policies in a simultaneous manner. That means, for the linearizing approach, that weights are considered simultaneously and implicitly [4], [8]. A recent proposal computes the Pareto set simultaneously without assuming a linearizing procedure or any other combination function [7].

The algorithm here presented also searches simultaneously for all the Pareto optimal deterministic solutions. Following the philosophy of Q-learning [15], the algorithm does not try to learn the model but just stores the Q-values.

To sum up, the main contribution of this paper is the presentation of a novel algorithm, MPQ-learning, (Multi-Pareto Q-learning) that approximates the set of all Pareto-optimal deterministic policies by directly generalizing Q-learning to the multiobjective setting.

This paper is structured as follows. Section 2 describes the necessary preliminaries, introduces the new algorithm and illustrates its behaviour with a simple example. Section 3 presents the results of the application of MPQ-learning to some benchmark problems, and compares its performance with that of a linearization procedure. Finally, some conclusions are drawn from the presented results.

2. MPQ-learning

MPQ-learning aims to simultaneously find all Pareto-optimal deterministic non-dominated policies for a MOMDP. In the following we describe some necessary concepts.

A MOMDP is defined by a set S of states, a set A of actions, a transition function $P : S \times A \times S \to [0, 1]$, where $P(s, a, s')$ is the probability of going from s to s' when executing a, and a reward function $R : S \times A \times S \to \mathbb{R}^n$, where vector $\vec{r} = R(s, a, s')$ is the expected immediate reward obtained is such case.

A *deterministic policy* is a function that, conditioning on the current state s (and possibly on the time step t if it is non-stationary), selects an action a. The execution of a deterministic policy from a given state s_0 (and possibly a time step t_0) leads to a sequence $s_0, \vec{r}_1, s_1, \vec{r}_2, \ldots, s_i, \vec{r}_{i+1} \ldots$ Given

a policy π, the *discounted accumulated return* is given by an *expected vector return* $E_\pi\{\vec{R}_t\} \in \mathbb{R}^n$, $E_\pi\{\vec{R}_t\} = E_\pi\{\Sigma_{k=0}^{\infty}\gamma^k \overrightarrow{r}_{t+k+1}\}$, where $E_\pi\{\}$ denotes the expected value given that the agent follows policy π.

Here we consider the *dominance relation* $\succ$ or *Pareto order* in $\mathbb{R}^n$. For every pair of vectors $\vec{v}, \vec{w} \in \mathbb{R}^n$, $\vec{v} \succ \vec{w}$ iff there exists a dimension i such that $v_i > w_i$ and there is no dimension j such that $v_j < w_j$. We write $\vec{v} \succeq \vec{w}$ when $\vec{v} \succ \vec{w}$ or $\vec{v} = \vec{w}$. Given a set $X \subset \mathbb{R}^n$, the *Pareto front* of X, $\mathrm{ND}(X)$, is the set of non-dominated vectors in X, that is, $\mathrm{ND}(X) = \{\vec{x} \in X \mid \nexists \vec{y} \in X \;\; \vec{y} \succ \vec{x}\}$.

As stated by [2], as we aim to learn the set of all non-dominated policies at once, we need an off-line algorithm like Q-learning, since the policy used to interact with the environment will not be the same that is learned. Indeed, we only can follow one policy per episode, and still need to improve several policies simultaneously. MPQ-learning is hence a direct extension of the scalar reinforcement learning technique Q-learning, that we cover in detail in the next section.

2.1. Q-learning

Q-learning [15] is a RL algorithm that learns a unique policy when rewards are scalar values. It learns scalar action-values $Q(s, a) : S \times A \rightarrow \mathbb{R}$, that represent the expected accumulated reward when following a given policy after taking a in s. The action a selected by the policy in each state is given by the expression $\mathrm{argmax}_a\, Q(s, a)$. The policy learned by Q-learning is thus stationary, since it only conditions on the current state s. In single-objective MDPs, there always exists a deterministic stationary optimal policy [5].

In Q-learning, a decision-making agent interacts with the environment through a sequence of steps. In the nth step, the agent observes its current state s_n, selects and performs an action a_n, observes the following state s', receives an immediate reward r_n and adjusts the value $Q(s_n, a_n)$ using a learning factor α_n. The updating expression of the scalar Q-learning algorithm is the following:

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n[r_n + \gamma V_{n-1}(s')] & \text{if } s = s_n \wedge a = a_n \\ Q_{n-1}(s,a) & \text{otherwise} \end{cases} \tag{1}$$

where

$$V_{n-1}(s) = \max_{a \in A} Q_{n-1}(s,a) \tag{2}$$

2.2. Updating expression of MPQ-learning

In this Section we provide the updating expression of MPQ-learning. Prior to defining this expression, we need to set some definitions.

The *vector value* of a state s under a policy π, denoted $\vec{v}^{\,\pi}(s)$, is the expected vector return when starting from s and following π. We call $\vec{v}^{\,\pi}(s)$ the *state-vector function* for policy π. The vector value of taking action a in state s under a policy π, denoted $\vec{q}^{\,\pi}(s,a)$ is the expected vector return when starting from s, taking action a, and thereafter following π. We call $\vec{q}^{\,\pi}(s,a)$ the *action-vector function* for policy π.

We will say that a policy π dominates or equals a policy π' iff $\forall s \in S$, $\vec{v}^{\,\pi}(s) \succeq \vec{v}^{\,\pi'}(s)$. A policy is non-dominated iff it is not dominated by any other possible policy. We denote a non-dominated policy as π^*, and its non-dominated action-vector function as $\vec{v}^{\,\pi*}(s)$.

We denote the set of all action-vectors of taking action a in state s under any non-dominated stationary policy π^* as $\mathbb{Q}^*(s,a)$.

Just like Q-learning, MPQ-learning learns action-vector values through repeated interaction with the environment. However, there are important differences that arise from the fact that the obtained rewards are vectors, and that the algorithm does not learn a single policy, but a set of policies at the same time.

The values learned in MPQ-learning are now sets $\mathbb{Q}(s,a)$ of vectors, which are used to estimate the optimal $\mathbb{Q}^*(s,a)$ sets. The essence of MPQ-learning is described by an updating expression, in many senses analogous to that described in Equation 1. Indeed, a direct approach would be to use Expression 1 as is, substituting the scalar operators with set operators when necessary:

$$\mathbb{Q}_n^{naive}(s,a) = \begin{cases} (1-\alpha_n)\mathbb{Q}_{n-1}^{naive}(s,a) \oplus \alpha_n[\vec{r}_n + \gamma \mathbb{V}_{n-1}^{naive}(s')] & \text{if } s = s_n \wedge a = a_n \\ \mathbb{Q}_{n-1}^{naive}(s,a) & \text{otherwise} \end{cases} \tag{3}$$

Where the definition of $\mathbb{V}^{naive}(s)$ is analogous to the one used by [16] and [17]:

$$\mathbb{V}^{naive}(s) = \mathrm{ND} \bigcup_{a \in A} \mathbb{Q}(s,a) \tag{4}$$

The operator $\oplus$ in Equation 3 performs a pairwise summation of the set corresponding to the old estimation set $\mathbb{Q}_{n-1}^{naive}(s,a)$ and the set corresponding to the new one $(\vec{r}_n + \gamma \mathbb{V}_{n-1}^{naive}(s'))$. This leads to an uncontrolled growth of

the sets: if we have two vectors in the old estimation set and two on the new one, we come up with four vectors in the updated set $\mathbb{Q}_n^{naive}(s,a)$, and after a few updating steps we would have an intractable number of vectors.

Instead, we want to establish a correspondence between the vectors of the old estimation set and the vectors of the new one, so if we have two vectors in the old set and two on the new one, then we will have at most two vectors inside the updated set. For this, we need to store some extra information along with the action-vectors inside the sets.

Formally, each *vector estimate* in $\mathbb{Q}(s,a)$ will consist of a pair $(\vec{q}, P)$,where $\vec{q}$ is the current value of the vector estimate, and P is a set of indices. The set P allows to control the learning process and also to exploit the learned policies once the learning process is completed. Each index $p \in P$ is a pair (s', i) where s' is an identifier of the accessed state and i stands for the i-th vector in $\mathbb{V}(s')$, precisely the one that is used to update $\vec{q}$. Notice that the dimension of $\vec{q}$, given by the number of objectives, is fixed and the same for every state. However, the size of P can be different for different states and grows during the execution of the algorithm from 0 to the number of states reachable from s after performing action a.

We will say that a pair (s', i) is new for a set $\mathbb{Q}(s,a)$ when there is no pair $(\vec{q}, P) \in \mathbb{Q}(s,a)$ such that $(s', i) \in P$, and write $(s', i) \not\sqsubset \mathbb{Q}(s,a)$. Otherwise we will say that (s', i) is not new in $\mathbb{Q}(s,a)$ and write $(s', i) \sqsubset \mathbb{Q}(s,a)$. We will say that a state s' is new for a set $\mathbb{Q}(s,a)$ when there is no index i such that $(s', i) \sqsubset \mathbb{Q}(s,a)$. We will write $s' \not\sqsubset \mathbb{Q}(s,a)$. Otherwise we will say that s' is not new in $\mathbb{Q}(s,a)$ and write $s' \sqsubset \mathbb{Q}(s,a)$. We also define the set $P \setminus s'$ as the result of removing the pair (s', i) from P, that is, $P \setminus \{(s', i)\}$ (whatever i is).

We will formally define the sets $\mathbb{V}(s)$ in function of the vector estimates:

$$\mathbb{V}(s) = \text{ND} \bigcup_{a \in A} \{\vec{q} \mid (\vec{q}, P) \in \mathbb{Q}(s,a)\} \tag{5}$$

We can identify two different kinds of operations in the updating process of $\mathbb{Q}(s,a)$,

1. Creating new vector estimates every time a state s' is reached for the first time after performing action a in state s.
2. Updating $\mathbb{Q}(s,a)$ after an already traversed transition leads to some node s'. This includes updating, creating, and deleting particular vector estimates.

We assume that for all $s \in S$ and for all $a \in A$, $\mathbb{Q}_0(s,a) = \{(\vec{0}, \varnothing)\}$. The updating expression for MPQ-learning is,

$$\mathbb{Q}_n(s,a) = \begin{cases} \mathbb{N}_{n-1}(s,a) \cup \mathbb{U}_{n-1}(s,a) \cup \mathbb{E}_{n-1}(s,a) & \text{if } s = s_n \wedge a = a_n \\ \mathbb{Q}_{n-1}(s,a) & \text{otherwise} \end{cases} \tag{6}$$

The updated Q-set comes from the union of three sets: $\mathbb{N}$, $\mathbb{U}$ and $\mathbb{E}$. The contents of these sets will be determined by the nature of the involved transition (s, a, s') in the current time step. Particularly, if the transition is *new* to the agent, i.e., it is the first time that s' is reached from s through action a, only the set $\mathbb{N}$ (standing for *new*) will contain vector estimates. However, if the transition has already been executed by the learning agent, then $\mathbb{N}$ will be empty and only the sets $\mathbb{U}$ and $\mathbb{E}$ can contain vector estimates. The set $\mathbb{U}$ (standing for *updated*) will contain vector estimates of $\mathbb{Q}_{n-1}(s,a)$ that had previously been linked to a vector estimate inside $\mathbb{V}_{n-1}(s')$, conveniently updated with this vector in $\mathbb{V}_{n-1}(s')$ and the obtained reward. The set $\mathbb{E}$ (standing for *extra*) will contain vector estimates if some extra vector estimate has appeared in $\mathbb{V}_{n-1}(s')$ since the last time the transition was visited by the agent. In the following we provide formalizations and more detailed explanations for these three sets.

When a state s' is reached for the first time from state s through action a, every vector estimate in $\mathbb{Q}_{n-1}(s,a)$ is going to be also inside $\mathbb{Q}_n(s,a)$, updated accordingly with the vector estimates inside $\mathbb{V}_{n-1}(s')$. If $\mathbb{V}_{n-1}(s')$ only has one vector estimate, then $|\mathbb{Q}_n(s,a)| = |\mathbb{Q}_{n-1}(s,a)|$. In general, when s' is reached for the first time, $|\mathbb{Q}_n(s,a)| = m|\mathbb{Q}_{n-1}(s,a)|$, where m is the number of vector estimates inside $\mathbb{V}_{n-1}(s')$. These vector estimates are stored inside the set $\mathbb{N}$, which is defined as follows:

$$\begin{aligned} \mathbb{N}_{n-1}(s,a) = \{((1-\alpha_n)\vec{q} + \alpha_n[\vec{r}_n + \gamma\vec{v}_j], P \cup \{(s', j)\} \mid \\ (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a) \wedge \vec{v}_j \in \mathbb{V}_{n-1}(s') \wedge s' \not\sqsubset \mathbb{Q}_{n-1}(s,a)\} \end{aligned} \tag{7}$$

When the reached state s' was already previously reached from state s through action a we can have two different situations. Set $\mathbb{U}$ deals with the vector estimates in $\mathbb{Q}(s,a)$ that had previously been updated with a vector in $\mathbb{V}(s')$. Set $\mathbb{E}$ deals with the extra vector estimates that have to appear inside $\mathbb{Q}(s,a)$ when a previously unknown vector appears in $\mathbb{V}(s')$.

The definition for the set $\mathbb{U}$ is somewhat analogous to the one in the single objective case:

$$\mathbb{U}_{n-1}(s,a) = \{((1-\alpha_n)\vec{q} + \alpha_n[\vec{r}_n + \gamma\vec{v}_j], P) \mid (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a) \wedge (s', j) \in P \wedge \vec{v}_j \in \mathbb{V}_{n-1}(s')\} \quad (8)$$

As we explained before, the set $\mathbb{E}$ will contain vector estimates when an extra vector $\vec{v}_j$ appears in $\mathbb{V}_{n-1}(s')$. The expression $s' \sqsubset \mathbb{Q}_{n-1}(s,a) \wedge (s', j) \not\sqsubset \mathbb{Q}_{n-1}(s,a)$ characterizes this situation: as this is not the first time s' is reached, $s' \sqsubset \mathbb{Q}_{n-1}(s,a)$, but as $\vec{v}_j$ is new, $(s', j) \not\sqsubset \mathbb{Q}_{n-1}(s,a)$. In this case, we cannot make use of vectors estimates inside $\mathbb{Q}_{n-1}(s,a)$ to determine the value of the new vector, because all of them have already been supported by some vector estimate inside previous instances of $\mathbb{V}(s')$, different from the extra vector $\vec{v}_j$. Hence, in this situation we have to start from the initial value $\vec{(0)}$ to determine the value of the vector in the new pair that is being inserted into $\mathbb{Q}_n(s,a)$. However, we can establish the new set of indices with the available information of the pairs inside $\mathbb{Q}_{n-1}(s,a)$. Given an extra vector $\vec{v}_j$, we will insert a new vector estimate in $\mathbb{Q}_n(s,a)$ for each vector estimate inside $\mathbb{Q}_{n-1}(s,a)$, with the set of indices that arises of removing the pair of the form (s', k) (whatever k is) from the set of indices of the vector estimate, and then adding the pair (s', j):

$$\mathbb{E}_{n-1}(s,a) = \{(\alpha_n[\vec{r}_n + \gamma\vec{v}_j], (P \setminus s') \cup \{(s', j)\}) \mid \vec{v}_j \in \mathbb{V}_{n-1}(s') \ \wedge \ s' \sqsubset \mathbb{Q}_{n-1}(s,a) \ \wedge \ (s', j) \not\sqsubset \mathbb{Q}_{n-1}(s,a) \ \wedge \ \exists \vec{q}\, (\vec{q}, P) \in \mathbb{Q}_{n-1}(s,a)\} \quad (9)$$

When a vector estimate that was present in previous instances of $\mathbb{V}(s')$ (and thus there is an associated vector estimate inside $\mathbb{Q}_{n-1}(s,a)$) has been removed, it is because now it represents a dominated policy. In that situation, the vector estimate inside $\mathbb{Q}_{n-1}(s,a)$ supported by the vector estimate removed from $\mathbb{V}_{n-1}(s')$ is not inside $\mathbb{Q}_n(s,a)$.

Notice that MPQ-learning keeps a vector in $\mathbb{Q}(s,a)$ for each possible combination of non-dominated vectors from the state-vector sets of reachable states. However, by application of Bellman's optimality principle, only the subset of non-dominated state vectors of s, $\mathbb{V}(s)$, can be used to support action vectors in other Q sets. This helps to keep the number of policies tracked by the algorithm under control.

2.3. Action Selection Mechanism

Like Q-learning, MPQ-learning is an off-policy technique, meaning that the policy that is followed during learning is not the same as the one learned. Indeed, this fact is even more evident in the multi-objective setting, as we are learning several policies at once, but the agent is only following one path.

This off-policy essence allows the agent to follow any policy during the learning process, as long as every state-action pair (s, a) is visited a potentially infinite number of times. Nevertheless, devising a proper action-selection strategy can lead to a better online performance.

The intuition behind our action-selection strategy is that, when exploiting the already acquired knowledge, the chances of choosing an action a, being in state s, are proportional to the number of vector estimates of $\mathbb{Q}(s, a)$ that are also inside $\mathbb{V}(s)$.

$$Pr\{a_n = a | s_n = s\} = \frac{|\{\vec{q} : \vec{q} \in \mathbb{V}_{n-1}(s) \wedge \exists P : (\vec{q}, P) \in \mathbb{Q}_{n-1}(s, a)\}|}{|\mathbb{V}_{n-1}(s)|} \tag{10}$$

In our experiments we have used a ϵ-greedy mechanism, that is, with probability ϵ a random action is chosen, and with $(1 - \epsilon)$ an action is chosen according to Expression 10.

2.4. Using the Learned Policies

In scalar Q-learning, once the values have been learned, we can derive the optimal (or near-optimal) policy by choosing the action that yields the maximum value. That is, if we are in state s, the chosen action is $argmax_a Q(s, a)$. However, in MPQ-learning we cannot use this operator, since we do not have scalar values $Q(s, a)$ anymore, but sets $\mathbb{Q}(s, a)$.

Once the learning process is complete, we can resort to a weight combination or just to a user selection process in which a single solution is directly chosen from the set of learned policies. Regardless of the selection method, the decision has to be made in the time step t_0, that is, at the beginning of the process.

The process for deriving a concrete learned policy in our case starts by computing the V-set of the initial state (see Expression 5). Once we have this V-set, we have to select a vector estimate out of it:

$$\vec{q}_{t_0} = \Theta(\mathbb{V}(s_0)) \tag{11}$$

Where $\Theta()$ is a selection operator. The action to be taken in state s_0 and time step t_0 is the one whose associated Q-set contains the chosen vector estimate:

$$\pi(s_0, t_0) = a : (\exists P_{t_0} : (\vec{q}_{t_0}, P_{t_0}) \in \mathbb{Q}(s_0, a)) \tag{12}$$

In the following time steps the selection operator is no longer needed, because the decision has been already done. We must resort to the set of indices P associated to the last chosen action, that is, P_{t_0}. In time step t_1, we need to look for the vector estimate inside $\mathbb{V}(s_1)$ indexed in P_{t_0}:

$$\vec{q}_{t_1} = \vec{v}_i : \vec{v}_i \in \mathbb{V}(s_1) \wedge (s_1, i) \in P_{t_0} \tag{13}$$

And the chosen action in state s_1 and time step t_1 is:

$$\pi(s_1, t_1) = a : (\exists P_{t_1} : (\vec{q}_{t_1}, P_{t_1}) \in \mathbb{Q}(s_1, a)) \tag{14}$$

It could happen that, when $n > 0$, we cannot find the suitable vector estimate $\vec{q}_{t_n}$ inside $\mathbb{V}(s_n)$. This is a symptom that the learning process is not completed, and we still have a vector estimate inside $\mathbb{V}(s_0)$ whose supporting values are no longer non-dominated. In that case we can use the selection operator again in order to choose a vector estimate inside $\mathbb{V}(s_n)$.

Thus, in general, if we are in state s_n at time step t_n, the action to take is given by the expression:

$$\pi(s_n, t_n) = a : (\exists P_{t_n} : (\vec{q}_{t_n}, P_{t_n}) \in \mathbb{Q}(s_n, a)) \tag{15}$$

Where

$$\vec{q}_{t_n} = \begin{cases} \Theta(\mathbb{V}(s_0)) & \text{if } n = 0 \\ \vec{v}_i : \vec{v}_i \in \mathbb{V}(s_n) \wedge (s, i) \in P_{t-1} & \text{if } n > 0 \wedge \exists \vec{v}_i \\ \Theta(\mathbb{V}(s_n)) & \text{otherwise} \end{cases} \tag{16}$$

2.5. Example

Let us assume a state transition diagram for a multi-objective Markov decision process as depicted in figure 1. Episodes always start at state s_1, and states s_3, s_4, s_5 are terminal. Action a_1 has a probabilistic outcome and may lead to states s_2 or s_3. Let us assume that for all terminal states s' we have $\mathbb{V}(s') = \{[\vec{v}_1 = (0, 0), \emptyset]\}$, i.e. each state has a single zero vector that is not supported by any index. Additionally, values for $\mathbb{Q}$ are not defined, since there are no available actions at terminal states.

We will apply MPQ-learning to this example assuming $\alpha = 0.1$, and $\gamma = 1$. The evolution of $\mathbb{Q}$ and $\mathbb{V}$ are displayed in table 1 for all state-action pairs and non-terminal states, and for each time step. The initial conditions are given for time step $t = 0$. What follows is a description of a possible sequence of transitions that illustrates the application of the MPQ-learning rule.

1. A first episode starts at s_1, where a_1 is the only action available. Let us assume the transition leads to s_2 with reward $\vec{r} = (0, 0)$. Since $\mathbb{V}(s_2)$ has two vectors, the application of the MPQ-learning rule results in the following,

$$\begin{aligned}
\mathbb{N}_1(s_1, a_1) &= \{[(0.9 \times (0, 0) + 0.1 \times ((0, 0) + (0, 0)), \{(s_2, 1)\}] \\
&\quad [(0.9 \times (0, 0) + 0.1 \times ((0, 0) + (0, 0)), \{(s_2, 2)\}]\} \\
\mathbb{U}_1(s_1, a_1) &= \varnothing \\
\mathbb{E}_1(s_1, a_1) &= \varnothing \\
\mathbb{Q}_1(s_1, a_1) &= \{[\vec{v}_1 = (0, 0), \{(s_2, 1)\}] \\
&\quad [\vec{v}_2 = (0, 0), \{(s_2, 2)\}]\}
\end{aligned}$$

2. Assume that at $t = 2$ action a_2 is chosen, leading to state s_4 with reward $\vec{r} = (1000, 2000)$. Notice that after this step, $\mathbb{V}(s_2)$ has only one non-dominated vector.

$$\begin{aligned}
\mathbb{N}_2(s_2, a_2) &= \{[(0.9 \times (0, 0) + 0.1 \times ((1000, 2000) + (0, 0)), \{(s_4, 1)\}]\} \\
\mathbb{U}_2(s_2, a_2) &= \varnothing \\
\mathbb{E}_2(s_2, a_2) &= \varnothing \\
\mathbb{Q}_2(s_2, a_2) &= \{[\vec{v}_1 = (100, 200), \{(s_4, 1)\}]\}
\end{aligned}$$

3. At $t = 3$ a new episode starts, transitioning again from s_1 to s_2. Now, $\mathbb{V}(s_2) = \{[\vec{v}_1 = (100, 200)(s_4, 1)]\}$, therefore,

$$
\begin{aligned}
\mathbb{N}_3(s_1, a_1) &= \varnothing \\
\mathbb{U}_3(s_1, a_1) &= \{[(0.9 \times (0,0) + 0.1 \times ((0,0) + (100, 200)), \{(s_2, 1)\}]\} \\
\mathbb{E}_3(s_1, a_1) &= \varnothing \\
\mathbb{Q}_3(s_1, a_1) &= \{[\vec{v}_1 = (10, 20), \{(s_2, 1)\}]\}
\end{aligned}
$$

4. Assume now that at $t = 4$ action a_3 is chosen, due to an exploration step. This leads to state s_5 with reward $(2000, 1000)$. Since this state is reached for the first time from state-action pair (s_2, a_3), the rule is applied as follows,

$$
\begin{aligned}
\mathbb{N}_4(s_2, a_3) &= \{[(0.9 \times (0,0) + 0.1 \times ((2000, 1000) + (0,0)), \{(s_5, 1)\}]\} \\
\mathbb{U}_4(s_2, a_3) &= \varnothing \\
\mathbb{E}_4(s_2, a_3) &= \varnothing \\
\mathbb{Q}_4(s_2, a_3) &= \{[\vec{v}_2 = (200, 100), \{(s_5, 1)\}]\}
\end{aligned}
$$

5. At $t = 5$ a third episode starts, transitioning once again stochastically from s_1 to s_2. However, now $\mathbb{V}(s_2)$ presents an *extra* vector. Therefore,

$$
\begin{aligned}
\mathbb{N}_5(s_1, a_1) &= \varnothing \\
\mathbb{U}_5(s_1, a_1) &= \{[(0.9 \times (10, 20) + 0.1 \times ((0,0) + (100, 200)), \{(s_2, 1)\}]\} \\
\mathbb{E}_5(s_1, a_1) &= \{[(0.1 \times ((0,0) + (200, 100)), \{(s_2, 2)\}]\} \\
\mathbb{Q}_5(s_1, a_1) &= \{[\vec{v}_1 = (19, 38), \{(s_5, 1)\}] \\
&\qquad [\vec{v}_3 = (20, 10), \{(s_2, 2)\}]\}
\end{aligned}
$$

6. Assume the episode terminates with a new transition from s_2 to s_4. The update rule is applied as follows,

$$
\begin{aligned}
\mathbb{N}_6(s_2, a_2) &= \varnothing \\
\mathbb{U}_6(s_2, a_2) &= \{[(0.9 \times (100, 200) + 0.1 \times ((1000, 2000) + (0,0)), \{(s_4, 1)\}]\} \\
\mathbb{E}_6(s_2, a_2) &= \varnothing \\
\mathbb{Q}_6(s_2, a_2) &= \{[\vec{v}_1 = (190, 380), \{(s_4, 1)\}]\}
\end{aligned}
$$

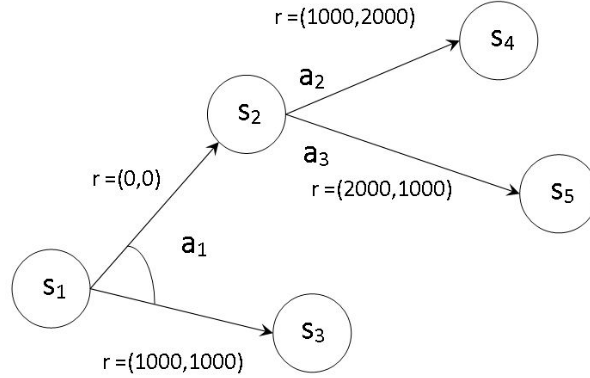


Figure 1: Sample state transition diagram.

7. Finally, let us assume the last episode leads from s_1 to s_3. This illustrates the case where each vector in $\mathbb{Q}(s_1, a_1)$ is supported by two vectors, one from s_2, and other from s_3,

$$\begin{aligned}
\mathbb{N}_7(s_1, a_1) &= \{[(0.9 \times (19, 38) + 0.1 \times ((1000, 1000) + (0, 0)), \{(s_2, 1)(s_3, 1)\}] \\
&\quad [(0.9 \times (20, 10) + 0.1 \times ((1000, 1000) + (0, 0)), \{(s_2, 2)(s_3, 1)\}]\} \\
\mathbb{U}_7(s_1, a_1) &= \varnothing \\
\mathbb{E}_7(s_1, a_1) &= \varnothing \\
\mathbb{Q}_7(s_1, a_1) &= \{[\vec{v}_1 = (117.1, 134.2), \{(s_2, 1)(s_3, 1)\}] \\
&\quad [\vec{v}_3 = (118, 109), \{(s_2, 2)(s_3, 1)\}]\}
\end{aligned}$$

3. Algorithm Comparison

In this section we perform an empirical comparison of MPQ-learning with the linear scalarization multi-policy approach, consisting in running Q-learning with several combinations of weights.

We use a dichotomic scalarizing procedure analogous to the one described in [1] to solve a test set of biobjective problems. We optimize a linear scalar function with a weights w_1 and w_2 associated to the first and second objectives respectively. First, we solve two instances of the problem using Q-learning, the first one with $w_1 >> w_2$, and the second one with $w_2 >> w_1$.

t	s	(s,a)	$\mathbb{Q}_t(s,a)$	$\mathbb{V}_t(s)$
0	s_1	(s_1,a_1)	$\vec{v}_1=(0,0)\ \emptyset$	$\vec{v}_1=(0,0)\ \emptyset$
	s_2	(s_2,a_2)	$\vec{v}_1=(0,0)\ \emptyset$	$\vec{v}_1=(0,0)\ \emptyset$
		(s_2,a_3)	$\vec{v}_2=(0,0)\ \emptyset$	$\vec{v}_2=(0,0)\ \emptyset$
1	s_1	(s_1,a_1)	$\vec{v}_1=(0,0)\ (s_2,1)$	$\vec{v}_1=(0,0)\ \ (s_2,1)$
			$\vec{v}_2=(0,0)\ (s_2,2)$	$\vec{v}_2=(0,0)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1=(0,0)\ \emptyset$	$\vec{v}_1=(0,0)\ \emptyset$
		(s_2,a_3)	$\vec{v}_2=(0,0)\ \emptyset$	$\vec{v}_2=(0,0)\ \emptyset$
2	s_1	(s_1,a_1)	$\vec{v}_1=(0,0)\ (s_2,1)$	$\vec{v}_1=(0,0)\ \ (s_2,1)$
			$\vec{v}_2=(0,0)\ (s_2,2)$	$\vec{v}_2=(0,0)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1=(100,200)\ (s_4,1)$	$\vec{v}_1=(100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(0,0)\ \emptyset$	
3	s_1	(s_1,a_1)	$\vec{v}_1=(10,20)\ (s_2,1)$	$\vec{v}_1=(10,20)\ \ (s_2,1)$
	s_2	(s_2,a_2)	$\vec{v}_1=(100,200)\ (s_4,1)$	$\vec{v}_1=(100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(0,0)\ \emptyset$	
4	s_1	(s_1,a_1)	$\vec{v}_1=(10,20)\ (s_2,1)$	$\vec{v}_1=(10,20)\ \ (s_2,1)$
	s_2	(s_2,a_2)	$\vec{v}_1=(100,200)\ (s_4,1)$	$\vec{v}_1=(100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(200,100)\ (s_5,1)$	$\vec{v}_2=(200,100)\ (s_5,1)$
5	s_1	(s_1,a_1)	$\vec{v}_1=(19,38)\ (s_2,1)$	$\vec{v}_1=(19,38)\ \ (s_2,1)$
			$\vec{v}_3=(20,10)\ (s_2,2)$	$\vec{v}_3=(20,10)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1=(100,200)\ (s_4,1)$	$\vec{v}_1=(100,200)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(200,100)\ (s_5,1)$	$\vec{v}_2=(200,100)\ (s_5,1)$
6	s_1	(s_1,a_1)	$\vec{v}_1=(19,38)\ (s_2,1)$	$\vec{v}_1=(19,38)\ \ (s_2,1)$
			$\vec{v}_3=(20,10)\ (s_2,2)$	$\vec{v}_3=(20,10)\ \ (s_2,2)$
	s_2	(s_2,a_2)	$\vec{v}_1=(190,380)\ (s_4,1)$	$\vec{v}_1=(190,380)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(200,100)\ (s_5,1)$	$\vec{v}_2=(200,100)\ (s_5,1)$
7	s_1	(s_1,a_1)	$\vec{v}_1=(117.1,134.2)\ \ (s_2,1)(s_3,1)$	$\vec{v}_1=(117.1,134.2)\ \ (s_2,1)(s_3,1)$
			$\vec{v}_3=(118,109)\ \ (s_2,2)(s_3,1)$	$\vec{v}_3=(118,109)\ \ (s_2,2)(s_3,1)$
	s_2	(s_2,a_2)	$\vec{v}_1=(190,380)\ (s_4,1)$	$\vec{v}_1=(190,380)\ (s_4,1)$
		(s_2,a_3)	$\vec{v}_2=(200,100)\ (s_5,1)$	$\vec{v}_2=(200,100)\ (s_5,1)$

Table 1: Evolution of estimated values over time with MPQ-learning. Thicker horizontal lines indicate division between training episodes.

Once the vector values associated to these two extreme supported solutions have been found, the procedure proceeds recursively. Given two different supported solution vectors $\vec{v}_A$ and $\vec{v}_B$, a new problem is defined assigning weights that match the slope defined by them. Solving the problem with Q-learning either results in one of the previous solutions, in which case we are done, or finds a new supported solution $\vec{v}_C$. In the latter case, two new problems need to be solved, considering the pairs of solutions $(\vec{v}_A, \vec{v}_C)$, and $(\vec{v}_C, \vec{v}_B)$ respectively. For a biobjective problem with $n > 1$ supported solutions, this procedure guarantees that all of them are found in $2n - 1$ scalar applications of Q-learning.

Performance has been measured according to the number of learning steps needed to reach the whole set of solutions inside $\mathbb{V}^*(s_0)$, where s_0 is the initial state of the environment. In the case of the scalarized procedure, this number is the sum of every individual search that has been performed.

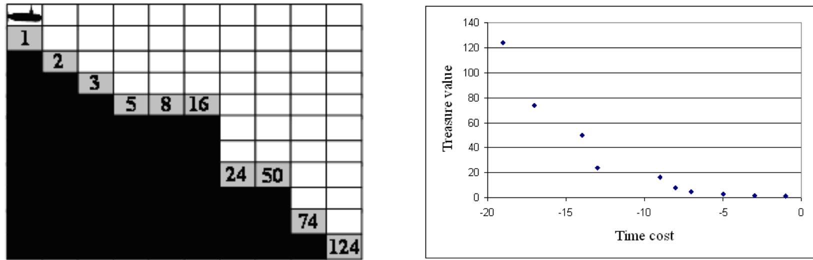
Notice that the number of reachable solutions varies according to the method that is being used, as linear scalarization techniques can only find supported solutions.

3.1. Deep Sea Treasure

We consider the *Deep Sea Treasure*(DST) environment in four different flavours. The original problem can be used to test if a MORL algorithm is able to find all the state vectors for a problem where these define a non-convex frontier in reward space [14]. DST has two objectives, and its true front for the selected initial state $\mathbb{V}^*(s_0)$, which is the top-left corner of the grid, contains ten state vectors.

The environment is a grid of 10 rows and 11 columns, as we can see in Figure 2(a). The agent controls a submarine that searches for undersea treasures. There are ten treasure locations with different values; the first objective is to minimize the time that the submarine takes to reach the treasure, and the second one is to maximize the value of the achieved treasure. The task is episodic, with each episode starting in the top-left position of the grid and ending when a treasure is reached, or after 1000 actions have been taken by the agent. At each step, four actions are available: moving one square to the top, right, bottom, or left. If an action would move the submarine outside of the grid, then the position of the submarine remains unchanged.

The reward received at each step is a vector of 2 elements; the first one is a punishment of -1 for the time consumed, and the second one is the



(a) DST Environment (reproduced from [14])

(b) DST Frontier (reproduced from [14])

Figure 2: Deep Sea Treasure problem: Environment (a) and Frontier (b)

value of the achieved treasure, that will be 0 in all steps except when the agent reaches a treasure location (the values are indicated in Figure 2(a)). In Figure 2(b) we can see the ten non-dominated vectors in $\mathbb{V}^*(s_i)$ associated to the non-dominated policies of this problem.

The original DST frontier only has two supported solutions out of the ten Pareto-optimal ones, and all the treasures are Pareto-optimal. In order to perform a more complete comparison, we have compared MPQ-learning and the scalarized version for three additional variants with different frontier configurations. In the following we explain the considered variants, and in Table 2 we gather the different features of their respective frontiers.

3.1.1. DST-2

In this second variant of DST we aim to test the algorithms behaviour when there are some non-optimal undersea treasures. Particularly, we have changed the value of the seventh treasure from 24 to 100, so treasures 50 and 74 are not Pareto-optimal any more.

3.1.2. DST-3

In the third variant of DST we want to test the algorithms behaviour when the number of non-supported solutions is closer to the number of Pareto-optimal solutions. Particularly, in this setting there are five supported solutions (as opposed to the two solutions in the original DST environment) out of ten Pareto-optimal solutions.

DST setting	Supported solutions	Pareto solutions
Original	2	10
DST-2	3	8
DST-3	5	10
DST-4	10	10

Table 2: Frontier features for the considered variants of the Deep Sea Treasure environment

3.1.3. DST-4

In this fourth setting of DST we want to test the algorithms behaviour when the number of non-supported solutions equals the number of Pareto-optimal solutions, that is, when the frontier is fully convex. In this variant there are ten supported solutions and ten Pareto-optimal solutions.

3.2. Results

For each setting we have executed 100 agents of the scalarized algorithm and 100 agents of MPQ-learning. The parameter setting for every execution has been the following:

- Discount rate $\gamma = 1$
- Learning rate $\alpha = 0.1$
- Exploration rate $\epsilon = 0.4$

In Table 3 we gather the average and maximum number of training steps that each method needs to converge to the whole set of reachable solutions in $\mathbb{V}^*(s_0)$. Recall that the scalarized scheme only reaches supported solutions.

3.3. Discussion

As we have already mentioned, it is well known that scalarizing methods, particularly those based on linear functions, can only obtain the convex hull, that is, the set of supported solutions of a MOMDP. On the contrary, MPQ-learning can yield the whole Pareto front.

It can be argued that in some cases it suffices to learn the set of supported solutions. However, in the light of the obtained results, even when only supported solutions are sought, in some cases MPQ-learning can outperform the linear scalarizing method in terms of training steps, as we can

Problem	Avg. steps	Max steps
Original (SCAL)	1074645.3	1494433
Original (MPQ)	1260451.1	3478320
DST-2 (SCAL)	14786851.0	23931695
DST-2 (MPQ)	23905464.7	39318415
DST-3 (SCAL)	5862617.5	10698936
DST-3 (MPQ)	4602536.4	6229090
DST-4 (SCAL)	15618235.0	19845973
DST-4 (MPQ)	6294062.1	10193190

Table 3: Scalarized algorithm vs. MPQ-learning: training steps until convergence of $V(0,0)$, over 100 agents

see in Table 3. Particularly, it has been the case of the problems DST-3 and DST-4. It seems reasonable to speculate that when the ratio of supported solutions/Pareto solutions grows, MPQ-learning tends to need less training steps than the linear scalarized method.

Notice that MPQ-learning computes in principle non stationary optimal policies. However, in the studied cases there are no such policies, so they do not appear in the final results. Moreover, the trace of the execution shows that they do not appear at any point of the execution.

4. Conclusions and Future Work

This work describes MPQ-learning, a temporal-difference method aimed at solving Multi-objective MDPs. MPQ-learning is based on Q-learning and differs from this scalar method in two main factors: the use of vector rewards, and the ability to learn the set of all non-dominated deterministic policies simultaneously. To our knowledge, this is the first temporal-difference method that tries to learn the whole set of non-dominated policies at the same time by means of a simultaneous updating of different value functions.

This approach can help to overcome some difficulties of other approaches that try to approximate the set of non-dominated policies with scalarizing functions. These resort in general to repeated calls to scalar reinforcement learning techniques with different preference configurations. MPQ-learning does not require these additional parameter adjustments, nor an explicit

scalarizing function, and is able to approximate convex as well as non-convex Pareto fronts.

We have compared MPQ-learning to a linear scalarizing algorithm based on running Q-learning several times with different preference configurations. In the light of the obtained results, in certain cases MPQ-learning can outperform the linear scalarizing method in terms of training steps. Hence, even when only supported solutions are sought, MPQ-learning can be a sensible option to solve a MOMDP.

The work described here can be refined in several ways. One of them is trying to avoid non-stationary policies in problems where they form part of the set of solutions, as well as testing the algorithm in stochastic environments. Evaluating this and other improvements, as well as completing a formal analysis of MPQ-learning, are important continuations of this work.

5. Acknowledgements

This work is partially funded by grants TIN2009-14179 (Spanish Government, Plan Nacional de I+D+i) and TIN2016-80774-R (AEI/FEDER, UE) (Spanish Government, Agencia Estatal de Investigación; and European Union, Fondo Europeo de Desarrollo Regional). Manuela Ruiz-Montiel is funded by the Spanish Ministry of Education through the National F.P.U. Program.

[1] Balachandran, M., Gero, J.S., 1984. A comparison of three methods for generating the pareto optimal set. Engineering Optimization 7, 319–336.

[2] Barrett, L., Narayanan, S., 2008. Learning all optimal policies with multiple criteria, ACM. pp. 41–47.

[3] Castelletti, A., Corani, G., Rizzolli, A., Soncini-Sessa, R., Weber, E., 2007. Reinforcement learning in the operational management of a water system, in: IFAC Workshop on Modeling and Control in Environmental Issues, pp. 325–330.

[4] Hiraoka, K., Yoshida, M., Mishima, T., 2008. Parallel reinforcement learning for weighted multi-criteria model with adaptive margin, in: Ishikawa, M., Doya, K., Miyamoto, H., Yamakawa, T. (Eds.), Neural Information Processing. volume 4984 of *Lecture Notes in Computer Science*, pp. 487–496.

[5] Howard, R.A., 1960. Dinamic programming and Markov Decision Processes. MIT Press.

[6] Lizotte, D.J., Bowling, M., Murphy, S.A., 2010. Efficient reinforcement learning with multiple reward functions for randomized controlled trial analysis, in: Proceedings of the 27th International Conference on Machine Learning, pp. 695–702.

[7] Moffaert, K.V., Nowé, A., 2014. Multi-objective reinforcement learning using sets of pareto dominating policies. Journal of Machine Learning Research 15, 3663–3692.

[8] Mukai, Y., Kuroe, Y., Iima, H., 2012. Multi-objective reinforcement learning method for acquiring all pareto optimal policies simultaneously, in: Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on, pp. 1917–1923.

[9] Natarajan, S., Tadepalli, P., 2005. Dynamic preferences in multi-criteria reinforcement learning, ACM. pp. 601–608.

[10] Roijers, D.M., Vamplew, P., Whiteson, S., Dazeley, R., 2013. A survey of multi-objective sequential decision-making. Journal of Artificial Intelligence Research (JAIR) 48, 67–113.

[11] Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., de-la Cruz, J.L.P., 2013. Design with shape grammars and reinforcement learning. Advanced Engineering Informatics 27, 230 – 245. URL: http://www.sciencedirect.com/science/article/pii/S1474034612001139, doi:10.1016/j.aei.2012.12.004.

[12] Vamplew, P., Dazeley, R., Barker, E., Kelarev, A., 2009. Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks, in: AI'09: The 22nd Autralasian Conference on Artificial Intelligence, pp. 340–349.

[13] Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., Dekker, E., 2011. Empirical evaluation methods for multiobjective reinforcement learning algorithms. Mach. Learn. 84, 51–80.

[14] Vamplew, P., Yearwood, J., Dazeley, R., Berry, A., 2008. On the limitations of scalarisation for multi-objective reinforcement learning of pareto

fronts, in: AI 2008: Advances in Artificial Intelligence. Springer. chapter 37, pp. 372–378.

[15] Watkins, C.J., 1989. Learning from delayed rewards. Ph.D. thesis. University of Cambridge.

[16] White, D.J., 1982. Multi-objective infinite-horizon discounted markov decision processes. Journal of Mathematical Analysis and Applications 89.

[17] Wiering, M.A., de Jong, E.D., 2007. Computing optimal stationary policies for Multi-Objective markov decision processes.