Computing Convex Coverage Sets for Multi-objective Coordination Graphs

Diederik M. Roijers[1], Shimon Whiteson[1], and Frans A. Oliehoek[2]

[1] Informatics Institute, University of Amsterdam, The Netherlands
{d.m.roijers,s.a.whiteson}@uva.nl
[2] Dept. of Knowledge Engineering, Maastricht University, The Netherlands
frans.oliehoek@maastrichtuniversity.nl

Abstract. Many real-world decision problems require making trade-offs between multiple objectives. However, in some cases, the relative importance of the objectives is not known when the problem is solved, precluding the use of single-objective methods. Instead, multi-objective methods, which compute the set of all potentially useful solutions, are required. This paper proposes new multi-objective algorithms for cooperative multi-agent settings. Following previous approaches, we exploit loose couplings, as expressed in graphical models, to coordinate efficiently. Existing methods, however, calculate only the *Pareto coverage set* (PCS), which we argue is inappropriate for stochastic strategies and unnecessarily large when the objectives are weighted in a linear fashion. In these cases, the typically much smaller *convex coverage set* (CCS) should be computed instead. A key insight of this paper is that, while computing the CCS is more expensive in unstructured problems, in many loosely coupled settings it is in fact cheaper to compute because the local solutions are more compact. We propose *convex multi-objective variable elimination*, which exploits this insight. We analyze its correctness and complexity and demonstrate empirically that it scales much better in the number of agents and objectives than alternatives that compute the PCS.

Keywords: Multi-agent systems, Multi-objective optimization, Game theory, Coordination graphs.

1 Introduction

In cooperative multi-agent systems, agents must coordinate their behavior in order to maximize their common utility. Key to making coordination efficient is exploiting the *loose couplings* common to such tasks: each agent's actions directly affect only a subset of the other agents. Such independence can be captured in a graphical model called a *coordination graph*, and exploited using methods such as *variable elimination* [8,9]. This paper considers how to address cooperative multi-agent systems in which the agents have multiple objectives, i.e., the utility is vector-valued. Many real-world problems have multiple objectives, e.g., maximizing performance of a computer network while minimizing power consumption [16].

The presence of multiple objectives does not in itself necessitate special solution methods. In many cases, the vector-valued utility function can be *scalarized*, i.e., converted to a scalar function. Subsequently, the original problem may be solvable with existing single-objective methods. However, this approach is not applicable when the

P. Perny, M. Pirlot, and A. Tsoukiàs (Eds.): ADT 2013, LNAI 8176, pp. 309–323, 2013.

parameters of the scalarization are not known in advance. For example, consider a company that produces different resources whose market prices vary. If there is not enough time to re-solve the decision problem for each price change, then we need multi-objective methods that compute a set of solutions optimal for all possible scalarizations.

This paper focuses on one-shot decision-making problems, for which several methods [5,6,12] have been developed. For instance Rollón [14] introduces an algorithm that we refer to as *multi-objective variable elimination* (MOVE), which solves multi-objective coordination graphs by iteratively solving local problems to eliminate agents from the graph. However, these methods all compute the *Pareto coverage set* (PCS), i.e., the Pareto front, of deterministic strategies.

In this paper, we argue that the PCS is often not the most appropriate solution concept. In the common case where the scalarization function is linear, the PCS is typically much larger than necessary. In addition, when joint strategies can be stochastic, the PCS is inadequate, even if the scalarization function is nonlinear.

To address these issues, we propose new methods that compute an alternative solution concept, the *convex coverage set* (CCS). The CCS is *the exact solution set* when the scalarization function is linear, and often much smaller than the PCS. In addition, it is a sufficient set of deterministic strategies from which to construct all optimal stochastic strategies. A key insight of this paper is that, while the CCS is more costly to compute than the PCS for nongraphical problems, it is often less costly to compute for loosely coupled problems because the *local* CCSs are much smaller than the local PCSs.

Thus, the main contribution of this paper is that it shows—both theoretically and empirically—that large speedups can be obtained when solving multi-objective coordination graphs by using the CCS as the solution concept. In particular, we 1) analytically show that the local CCSs can be much smaller than local PCSs, 2) present *convex MOVE* (CMOVE), an extension to MOVE that efficiently computes the CCS, 3) analyze the correctness and complexity of CMOVE in terms of the size of the coverage sets, and 4) demonstrate empirically that CMOVE scales much better than previous algorithms.[1]

2 Multi-objective Coordination Graphs

We formalize our problem setting as a multi-objective extension to coordination graphs [8]. In particular, a *multi-objective coordination graph* (MO-CoG) is a tuple $\langle \mathcal{D}, \mathcal{A}, \mathcal{U} \rangle$: $\mathcal{D} = \{1, ..., n\}$ is the set of n agents; $\mathcal{A} = \mathcal{A}_i \times ... \times \mathcal{A}_n$ is the joint action space (the Cartesian product of the finite action spaces of all agents) and $\mathcal{U} = \left\{\mathbf{u}^1, ..., \mathbf{u}^\rho\right\}$ is the set of ρ, d-dimensional *local payoff functions*. The total team payoff is the (vector) sum of local payoffs, with a limited scope e, i.e., the subset of agents that participate in it: $\mathbf{u}(\mathbf{a}) = \sum_{e=1}^{\rho} \mathbf{u}^e(\mathbf{a}_e)$. We use u_i to indicate the value of the i-th objective.

A team strategy π is a probability distribution over joint actions $\mathcal{A} \rightarrow [0,1]$. In general strategies are stochastic. Every *joint* action gets assigned a probability $0 \leq \pi(\mathbf{a}) \leq 1$, and the probabilities for all joint actions together sum to 1, $\sum_{\mathbf{a} \in \mathcal{A}} \pi(\mathbf{a}) = 1$. The value of a strategy $\mathbf{u}^\pi$ is the expected (vector-valued) utility of the strategy $\mathbf{u}^\pi = \sum_{\mathbf{a} \in \mathcal{A}} \pi(\mathbf{a}) \mathbf{u}(\mathbf{a})$. A deterministic strategy is a special case of a strategy in which one

[1] A preliminary version of this work was presented in [13].

joint action $\mathbf{a}$ has probability 1 and the rest probability 0. We refer to the set of all vectors for all possible strategies as $\mathcal{V}$.[2]

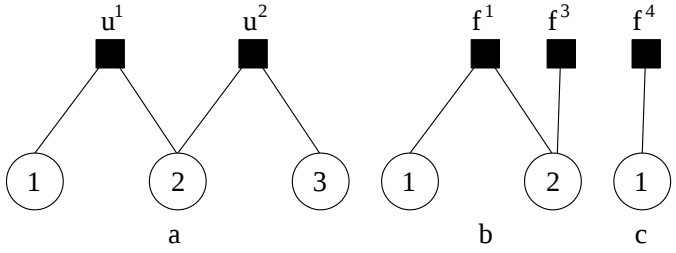

Fig. 1. (a) A MO-CoG factor graph, (b) after eliminating agent 3 by adding f^3, and (c) after eliminating agent 2 by adding f^4

The decomposition of $\mathbf{u}(\mathbf{a})$ into local payoff functions can be represented as a *factor graph* containing agents (variables) and local payoff functions (factors), with edges connecting local payoff functions to the agents in their scope. Figure 1a shows a factor graph for the payoff function $\mathbf{u}(\mathbf{a}) = \sum_{e=1}^{\rho} \mathbf{u}^e(\mathbf{a}_e) = \mathbf{u}^1(a_1, a_2) + \mathbf{u}^2(a_2, a_3)$.

We assume there exists a *scalarization function* f that converts $\mathbf{u}^\pi$ to a scalar payoff $u^\pi_{\mathbf{w}} = f(\mathbf{u}^\pi, \mathbf{w})$. This function is parameterized by a weight vector $\mathbf{w}$, which is unknown *when the MO-CoG is solved* but known when the agents must select a strategy. The solution to a MO-CoG is the *coverage set* (CS) [2], i.e., all strategies π and associated values $\mathbf{u}^\pi$ that are optimal for some $\mathbf{w}$:

$$CS(\mathcal{V}) = \left\{ \mathbf{u}^\pi : \mathbf{u}^\pi \in \mathcal{V} \ \wedge \ \exists \mathbf{w} \forall \pi' \ u^\pi_{\mathbf{w}} \geq u^{\pi'}_{\mathbf{w}} \right\}.$$

For convenience, we assume that the coverage set contains both the values and associated strategies. What the CS looks like depends on what strategies are allowed, and what we know about the scalarization function.

A minimal assumption about the scalarization function is that it is monotonically increasing in all objectives (i.e., if the value for one objective increases while the values for the other objectives stay constant, the scalarized value cannot go down). This assumption ensures that objectives are actually objectives, i.e., having more of them is better. In this case, the CS is called the *Pareto coverage set* (PCS) or *Pareto front*:

$$PCS(\mathcal{V}) = \left\{ \mathbf{u}^\pi : \mathbf{u}^\pi \in \mathcal{V} \ \wedge \ \neg \exists \pi' \ \mathbf{u}^{\pi'} \succ_P \mathbf{u}^\pi \right\},$$

where $\succ_P$ indicates *Pareto dominance* (P-dominance): greater or equal in all objectives and strictly greater in at least one objective. Note that computing P-dominance[3] requires only comparing pairs of vectors [7].

A highly prevalent scenario is that, in addition to knowing that the scalarization function is monotonically increasing, we also know that it is linear, $f = \mathbf{w} \cdot \mathbf{u}^\pi$. This is

[2] MO-CoGs are similar to the *multi-objective weighted constraint satisfaction problems* (MO-WCSPs) considered in [15]. However, MO-WCSPs consider only deterministic strategies and bounded, integer-valued payoffs. In addition, they consider *constraints*, the absence of which in Mo-CoGs has important implications for our complexity analysis (see Section 6).

[3] P-dominance is often called *pairwise dominance* in the POMDP literature.

the case in, e.g., clinical trials [11] or resource gathering [1]. In this case, all we need is the convex coverage set (CCS):[4]

$$CCS(\mathcal{V}) = \left\{ \mathbf{u}^{\pi} : \mathbf{u}^{\pi} \in \mathcal{V} \ \wedge \ \exists \mathbf{w} \forall \pi' \ \mathbf{w} \cdot \mathbf{u}^{\pi} \geq \mathbf{w} \cdot \mathbf{u}^{\pi'} \right\}.$$

Vectors not in the CCS are $C\text{-}dominated$. In contrast to P-domination, C-domination cannot be tested for with pairwise comparisons because it can (in the setting of deterministic strategies) take two or more vectors to C-dominate a vector: a vector can be dominated over the entire weight-space, but not necessarily always by the same vector, as indicated in Figure 2 (right). The CCS contains all strategies that could be optimal for some weight in a linear scalarization, i.e., all strategies that are not C-dominated. Anything in the PCS but not in the CCS is C-dominated and cannot be useful given the assumption of a linear scalarization function. Because we assume the linear scalarization is monotonically increasing, we can represent it without loss of generality as a convex combination of the objectives: i.e., the weights are positive and sum to 1. Since such linear functions are a subset of monotonically increasing functions, the CCS is a subset of the PCS.

Many multi-objective methods, e.g., [5,6,12,14] simply assume that the PCS is the appropriate solution set. However, which CS one should use depends what one can assume about how utility is defined with respect to the multiple objectives, i.e., which scalarization function is used to scalarize the vector-valued payoffs. We argue that in many situations, one can assume that the scalarization function will be linear. For example, when the different objectives are products and/or resources that need to be bought and sold on a market, every objective will be associated with a current unit price on the market, leading to linear trade-offs. In such cases one should use the CCS.

In addition, the choice of solution concept also depends on whether only deterministic strategies are considered or whether stochastic ones are also permitted. We consider this issue in the next section.

3 Deterministic versus Stochastic Strategies

When we allow only deterministic strategies, i.e., one joint action is chosen with probability 1, the PCS and CCS can be quite different. In Figure 2 (left) the values of deterministic strategies are represented as points in value-space, for a two-objective MO-CoG. The strategy A is in both the CCS and the PCS. B, however, is in the PCS, but not the CCS, because there is no weight for which a linear scalarization of B's value would be optimal, as shown in Figure 2 (right), where the scalarized value of the strategies are plotted as a function of the weight on the first objective ($w_2 = 1 - w_1$). C is in neither the CCS nor the PCS: it is Pareto-dominated by A. We refer to the deterministic PCS as the PCS of deterministic strategies, i.e., the PCS when only deterministic strategies are allowed. We refer similarly to the deterministic CCS.

As discussed in Section 2, stochastic strategies are linear combinations of deterministic strategies. The value of a stochastic strategy is thus also a linear combination of the

[4] The convex coverage set is often called the *convex hull*. We avoid this term because it is imprecise: the convex hull (a term from graphics) is a superset of the convex coverage set.

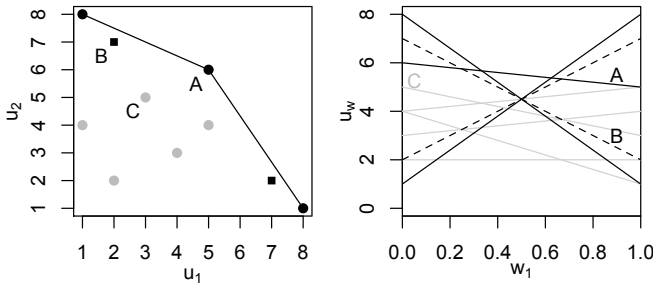

Fig. 2. The CCS (filled circles at left, and solid black lines at right) versus the PCS (filled circles and squares at left, and both dashed and solid black lines at right) for twelve random 2-dimensional payoff vectors

value vectors of the deterministic strategies it is a mixture of: $\mathbf{u}^{\pi} = \sum_{\mathbf{a} \in \mathcal{A}} \pi(\mathbf{a})\mathbf{u}(\mathbf{a})$. Therefore, the optimal values (for both linear and nonlinear monotonically increasing scalarization functions [17]) lay on the convex upper surface spanned by the strategies in the deterministic CCS, as indicated by the black lines in Figure 2 (left). In the stochastic case, the PCS and CCS are thus identical. Furthermore, the values for the stochastic PCS/CCS can be constructed from the values in the deterministic CCS. The stochastic PCS/CCS is thus very different from the deterministic PCS and the deterministic CCS. While the deterministic PCS and deterministic CCS contain finite numbers of strategies, the stochastic PCS/CCS contains inifinitely many strategies.

However, when we know that the scalarization function is linear, we do not actually need the entire stochastic CCS: for each weight, there exists a deterministic strategy that is optimal. For every optimal strategy in the stochastic CCS there exists a deterministic strategy that is just as good, because a linear combination of the values of two or more deterministic strategies never yields a larger scalarized utility for any $\mathbf{w}$, than one of the constituent deterministic strategies: $\mathbf{w} \cdot \mathbf{u}^{\pi} = \sum_{\mathbf{a}} \pi(\mathbf{a})(\mathbf{w} \cdot \mathbf{u}(\mathbf{a}))$. By contrast, when the scalarization function is monotonically increasing (but not necessarily linear), the full stochastic PCS is required. This is a problem, because it contains infinitely many strategies. However, all values on the stochastic PCS can be attained by making a stochastic mixture from the strategies on the deterministic CCS [17]. Note that these mixtures (all points on the black lines in Figure 2 (left)) dominate all points, like B, that are in the deterministic PCS but not the deterministic CCS. Therefore the CCS can be used to create all possible values on the PCS of stochastic strategies, and is more compact than the deterministic PCS.

It might of course be the case that the problem setting is restricted to deterministic solutions. For example in the medical domain [11], it can be unethical to treat patients based on a stochastic strategy. However, in most settings, stochasticity is permissable and the aim is to optimize the expected return.

Therefore, in this paper we present methods for computing the strategies in the deterministic CCS because it is an appropriate solution concept, not only when the scalarization function is linear, but also any time stochastic strategies are considered, even if the scalarization function is nonlinear, as shown in Table 1. For brevity, in the rest of the paper, we refer to the deterministic CCS as simply the CCS, to deterministic strategies as *joint actions*, and to the set of values of all deterministic strategies as $\mathcal{V}$.

Table 1. Motivating scenarios

	Linear scalarization functions	**Monotonically increasing scalarization functions**
Deterministic strategies	Deterministic CCS	Deterministic PCS
Stochastic strategies	Deterministic CCS	Deterministic CCS

4 Nongraphical Convex Approach

One way to compute the CCS, is to *ignore the graphical structure*, calculate the set of all possible payoffs for all joint actions $\mathcal{V}$, and prune away the C-dominated joint actions. To determine the set $\mathcal{V}$, we first translate the problem to a set of *value set factors* (VSFs), $\mathcal{F}$. Each VSF f is a function mapping local joint actions to sets of payoff vectors. Initially, the VSFs are constructed from the local payoff functions such that $f^e(\mathbf{a}_e) = \{\mathbf{u}^e(\mathbf{a}_e)\}$, i.e., each VSF maps a local joint action to the singleton set containing only that action's local payoff. We can now define $\mathcal{V}$ in terms of $\mathcal{F}$ using the *cross-sum* operator over all VSFs in $\mathcal{F}$ for each joint action $\mathbf{a}$: $\mathcal{V}(\mathcal{F}) = \bigcup_{\mathbf{a}} \bigoplus_{f^e \in \mathcal{F}} f^e(\mathbf{a}_e)$.[5] The CCS can now be calculated by applying a pruning operator `CPrune` (described below) that removes all C-dominated vectors from a set of value vectors, to $\mathcal{V}$:

$$CCS(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\bigcup_{\mathbf{a}} \bigoplus_{f^e \in \mathcal{F}} f^e(\mathbf{a}_e))$$

The CCS contains the all the vectors that are maximizing for some $\mathbf{w}$:

$$\forall \mathbf{a} \quad \left(\exists \mathbf{w}\ s.t.\ \mathbf{a} = \arg\max_{\mathbf{a} \in \mathcal{A}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a})\right) \implies \mathbf{u}(\mathbf{a}) \in CCS(\mathcal{V}(\mathcal{F})) \tag{1}$$

This is exactly the same problem as in *partially observable Markov decision processes* (POMDPs) [7], where the optimal α-vectors (corresponding to the value vectors $\mathbf{u}^\pi$) for all beliefs (corresponding to the weight vectors $\mathbf{w}$) must be found. Therefore, we can use pruning operators from the POMDP literature. Algorithm 1 describes our implementation of `CPrune`, which is based on [7] with the modification that, in order to improve runtime guarantees, we first pre-prune to the PCS using the `PPrune` operator shown in Algorithm 2, which computes the (deterministic) PCS in $O(d|\mathcal{V}_{det}||PCS|)$ by running pairwise comparisons.

Next, we maintain a partial CCS ($\mathcal{U}^*$), which is constructed as follows: we select a random vector $\mathbf{u}$ from the set of candidate vectors $\mathcal{U}'$ and test whether there is a weight vector $\mathbf{w}$ for which it is better than the vectors in $\mathcal{U}^*$ by solving the linear program shown in Algorithm 3. If so, we find the best vector $\mathbf{v}$ for $\mathbf{w}$ in $\mathcal{U}'$ and move $\mathbf{v}$ to $\mathcal{U}^*$. If there is no weight for which $\mathbf{u}$ is better, we remove $\mathbf{u}$ from $\mathcal{U}'$ (because it is C-dominated).

The runtime of the `CPrune` operator we use is $O(d|\mathcal{V}_{det}||PCS| + |PCS|P(d|CCS|))$, where $P(d|CCS|)$ is a polynomial in the size of the CCS and the number

[5] The cross-sum of two sets A and B contains all possible vectors that can be made by summing one payoff vector from each set: $A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A \wedge \mathbf{b} \in B\}$.

of objectives d, which is the runtime of the linear program that tests for C-domination (Algorithm 3).

Algorithm 1. `CPrune`$(\mathcal{U})$

$\mathcal{U}' =$ `PPrune`$(\mathcal{U})$
$\mathcal{U}^* = \emptyset$
while `notEmpty`$(\mathcal{U}')$ **do**
  select random $\mathbf{u}$ from $\mathcal{U}'$
  $\mathbf{w} \leftarrow$ `findWeight`$(\mathbf{u}, \mathcal{U}^*)$
  if $\mathbf{w}$=*null* **then**
    remove $\mathbf{u}$ from $\mathcal{U}'$
  else
    move best $\mathbf{v}$ for weight $\mathbf{w}$ from $\mathcal{U}'$ to $\mathcal{U}^*$
return $\mathcal{U}^*$

Algorithm 2. `PPrune`$(\mathcal{U})$

$\mathcal{U}^* \leftarrow \emptyset$
while $\mathcal{U} \neq \emptyset$ **do**
  $\mathbf{u} \leftarrow$ the first element of $\mathcal{U}$
  foreach $\mathbf{v} \in \mathcal{U}$ **do**
    if $\mathbf{v} \succ_P \mathbf{u}$ **then**
      $\mathbf{u} \leftarrow \mathbf{v}$ *// Continue with* $\mathbf{v}$ *instead of* $\mathbf{u}$
  Remove $\mathbf{u}$, and all vectors Pareto-dominated by it, from $\mathcal{U}$
  Add $\mathbf{u}$ to $\mathcal{U}^*$
return $\mathcal{U}^*$

5 Exploiting Loose Couplings

In the previous section, we showed that, for the nongraphical approach, computing the CCS is more expensive than computing the PCS. In this section, we show that, by exploiting the MO-CoG's graphical structure, we can often compute the CCS much more efficiently. In particular, we solve the MO-CoG as a series of local subproblems, by iteratively *eliminating* agents, and thereby manipulating $\mathcal{F}$. The key idea is, for each agent elimination, to compute a *local CCS* (LCCS), pruning away as many vectors as possible at the lowest possible level. This minimizes the number of payoff vectors that are calculated at the global level, which can greatly speed computation. Here we describe the `elim` operator for eliminating agents used by CMOVE in Section 6.

To eliminate agent i, we define $\mathcal{F}_i$, the set of relevant VSFs with i in scope. Then, for each possible local joint action of n_i, agent i's neighbors, we define an LCCS that contains the payoffs of the C-undominated responses of agent i to the given local joint action of n_i. In other words, it is the CCS of the subproblem that arises when considering only $\mathcal{F}_i$ and fixing a specific local joint action of n_i. To compute the LCCS, we must consider all payoff vectors and prune the dominated ones. If we fix all actions in $\mathbf{a}_{n_i}$ except a_i, the set of all payoff vectors for this subproblem is: $\mathcal{V}_i(\mathcal{F}_i, \mathbf{a}_{n_i}) = \bigcup_{a_i} \bigoplus_{f^e \in \mathcal{F}_i} f^e(\mathbf{a}_e)$, where $\mathbf{a}_e$ is formed from a_i and the appropriate part of $\mathbf{a}_{n_i}$. The corresponding LCCS is thus the undominated subset of $\mathcal{V}_i(\mathcal{F}_i, \mathbf{a}_{n_i})$:

$$LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i}) = CCS(\mathcal{V}_i(\mathcal{F}_i, \mathbf{a}_{n_i})).$$

Using these LCCSs we can define a new VSF, f^{new} conditioned on the actions of the agents in n_i: $\forall \mathbf{a}_{n_i}\ f^{new}(\mathbf{a}_{n_i}) \triangleq LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i})$. The `elim` operator replaces the VSFs in $\mathcal{F}_i$ in $\mathcal{F}$ by this new factor:

$$\texttt{elim}(\mathcal{F},i) = (\mathcal{F} \setminus \mathcal{F}_i) \cup \{f^{new}(\mathbf{a}_{n_i})\}.$$

Theorem 1. `elim` *preserves the CCS:* $\forall i\ \forall \mathcal{F}\ CCS(\mathcal{V}(\mathcal{F})) = CCS(\mathcal{V}(\texttt{elim}(\mathcal{F},i)))$.

Proof sketch. The linear scalarization function distributes over the local payoff functions: $\mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \mathbf{w} \cdot \sum_e \mathbf{u}^e(\mathbf{a}_e) = \sum_e \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e)$. Thus, when eliminating agent

Algorithm 3. `findWeight`$(\mathbf{u}, \mathcal{U})$

$$\max_{x,\mathbf{w}} \; x$$
$$\text{subject to } \; \mathbf{w} \cdot (\mathbf{u} - \mathbf{u}') - x \geq 0, \forall \mathbf{u}' \in \mathcal{U}$$
$$\sum_{i=1}^{d} w_i = 1$$

if $\mathbf{x} > \mathbf{0}$ **return** $\mathbf{w}$ **else return** null

Algorithm 4. `elim`$(\mathcal{F}, i,$ `prune1`, `prune2`$)$

$\mathcal{U}^*, n_i \leftarrow \emptyset$, set of neighboring agents of i
$\mathcal{F}_i \leftarrow$ the subset of f functions involving i
$f^{new}(\mathbf{a}_{n_i}) \leftarrow$ a new factor
foreach $\mathbf{a}_{n_i} \in \mathcal{A}_{n_i}$ **do**
  $f^{new}(\mathbf{a}_{n_i}) \leftarrow LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i},$ `prune1`, `prune2`$)$.
$\mathcal{F} \leftarrow \mathcal{F} \setminus \mathcal{F}_i \cup \{f^{new}\}$
return $\mathcal{V}^*$

i, we divide the set of VSFs into non-neighbors (nn), in which agent i does not participate, and neighbors (n_i) such that: $\mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \sum_{e \in nn} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e) + \sum_{e \in ni} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e)$. Now, following (1), the CCS contains $\max_{\mathbf{a} \in \mathcal{A}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a})$ for all $\mathbf{w}$. `elim` pushes this maximization in: $\max_{\mathbf{a} \in \mathcal{A}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \max_{\mathbf{a}_{-i} \in \mathcal{A}_{-i}} \sum_{e \in nn} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e) + \max_{a_i \in \mathcal{A}_i} \sum_{e \in ni} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e)$. `elim` replaces the agent-i factors by a term $f^{new}(\mathbf{a}_{n_i})$ that satisfies $\mathbf{w} \cdot f^{new}(\mathbf{a}_{n_i}) = \max_{a_i} \sum_{e \in n_i} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e)$ per definition, thus preserving the maximum for all $\mathbf{w}$ and thereby preserving the CCS.

Since LCCS $\subseteq$ LPCS $\subseteq \mathcal{V}_i$, where LPCS is the local PCS, `elim` not only reduces the problem size, it can do so more than is possible when considering only P-dominance. Consequently, focusing on the CCS can lead to considerable speedups.

6 Convex MOVE

We now present *Convex Multi-Objective Variable Elimination* (CMOVE), which implements `elim` using pruning operators, iteratively applies it to compute the CCS, and outputs the correct joint actions for each payoff vector in the CCS. It is an extension to Rollón's Pareto-based MOVE (which we denote PMOVE) [14].

Like PMOVE, CMOVE eliminates agents in sequence, solving local subproblems along the way. The most important difference is that CMOVE computes the CCS, which can lead to smaller subproblems and thus much better computational efficiency. In addition, we identify three places where pruning can take place, yielding a more flexible algorithm with different trade-offs. Finally, we use a *tagging scheme* instead of the *backwards pass* employed by Rollón, which greatly simplifies the algorithm without effecting its runtime.

CMOVE is also related to multi-objective methods for GAI networks [6] and influence diagrams [12]. However, like PMOVE, these methods compute only the PCS.

6.1 Algorithm

We first present an abstract version of CMOVE, which leaves the pruning operators unspecified. The choice of these operators leads to specific variants with different trade-offs between pruning effort and local problem sizes. As before, CMOVE first translates the problem into a set of *vector-set factors* (VSFs), $\mathcal{F}$. Next, it iteratively eliminates agents using `elim`. The elimination order can be determined using techniques devised for regular VE [10]. Algorithm 4 shows our implementation of `elim`, parameterized

with two pruning operators, `prune1` and `prune2`, corresponding to two different pruning locations inside $LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2})$, which is implemented as follows. First we define a new cross-sum-and-prune operator $A \hat{\oplus} B = \texttt{prune1}(A \oplus B)$, which we can apply sequentially in the definition of the LCCS operator:

$$LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2}) = \texttt{prune2}(\bigcup_{a_i} \hat{\bigoplus}_{f^e \in \mathcal{F}_i} f^e(\mathbf{a}_e)).$$

Applying `prune1` to each cross-sum of two sets, via the $\hat{\oplus}$ operator, leads to *incremental pruning* [4]; `prune2` prunes at a coarser level, after the union.

CMOVE applies `elim` iteratively until no agents remain, resulting in the CCS. An example of how this works is presented in Section 6.3.

Pruning can also be applied at the very end, after all agents have been eliminated, which we call `prune3`. In increasing level of coarseness, we thus have three pruning operators: incremental pruning (`prune1`), pruning after the union over actions of the eliminated agent (`prune2`), and pruning after all agents have been eliminated (`prune3`).

There are several ways to implement the pruning operators that lead to correct instantiations of CMOVE. One can use both `PPrune` (Algorithm 2) as well as `CPrune` (Algorithm 1) as long as either `prune2` or `prune3` is `CPrune`. (Note that if `prune2` computes the CCS, `prune3` is not necessary.) In this paper, we consider *Basic CMOVE*, which does not use `prune1` and `prune3` and only prunes at `prune2` using `CPrune`, as well as *Incremental CMOVE*, which uses `CPrune` at both `prune1` and `prune2`.

6.2 Tagging Scheme

Once CMOVE computes the CCS, we need to retrieve the joint actions that generate these values. In single-objective VE, this is typically done with a *backwards pass* that constructs a joint action by iterating through the eliminated agents in reverse order. However, doing so in the multi-objective setting is more complex, because the partial joint actions in the LCCSs need to be matched with the different values in the CCS instead of just backtracking a single optimal solution that automatically belongs to the optimal value. Consequently, the backwards pass used in Rollón's implementation of PMOVE [14] is fairly complex. However, we can obviate the need for a backwards pass by using a *tagging* scheme: when eliminating an agent i, CMOVE tags all the vectors in the LCCSs with the appropriate action of this agent. The payoff vectors are stored as a tuple containing both the payoff vector and a partial joint action. CMOVE combines the tags of agent i with the tags already present in $\mathcal{F}_i$. For example, in Figure 1, factor f^3 contains payoff vectors tagged with an action of agent 3 and factor f^4 contains tags with actions of both agents 2 and 3. Doing this for every agent in the elimination sequence builds the complete joint action for each payoff vector in the CCS. Replacing the backwards pass with this tagging scheme reduces by about half the number of lines of pseudocode needed to describe the algorithm.

6.3 Example

Consider the example in Figure 1a, using the payoffs defined by Table 2. First, CMOVE creates the VSFs f^1 and f^2 from u^1 and u^2 (not shown). To eliminate agent 3, it creates a new factor $f^3(a_2)$ by computing the LCCSs for every a_2 and tagging each element

of each set with the action of agent 3 that generates it. For $\dot{a}_2$, CMOVE first generates the set $\{(3,1)_{\dot{a}_3}, (1,3)_{\bar{a}_3}\}$. Since both of these vectors are optimal for some $\mathbf{w}$, neither is removed by pruning and thus $f^3(\dot{a}_2) = \{(3,1)_{\dot{a}_3}, (1,3)_{\bar{a}_3}\}$. For $\bar{a}_2$, CMOVE first generates $\{(0,0)_{\dot{a}_3}, (1,1)_{\bar{a}_3}\}$. `CPrune` determines that $(0,0)_{\dot{a}_3}$ is dominated and consequently removes it, yielding $f^3(\dot{a}_2) = \{(1,1)_{\bar{a}_3}\}$. CMOVE then adds f^3 to the graph and removes f^2 and agent 3, yielding the factor graph shown in Figure 1b.

CMOVE then eliminates agent 2 by combining f^1 and f^3 to create f^4. For $f^4(\dot{a}_1)$, CMOVE must calculate the LCCS of:

$$(f^1(\dot{a}_1,\dot{a}_2) \oplus f^3(\dot{a}_2)) \cup (f^1(\dot{a}_1,\bar{a}_2) \oplus f^3(\bar{a}_2)).$$

The first cross sum is $\{(7,2)_{\dot{a}_2\dot{a}_3}, (5,4)_{\dot{a}_2\bar{a}_3}\}$ and the second is $\{(1,1)_{\bar{a}_2\bar{a}_3}\}$. Pruning their union yields $f^4(\dot{a}_1) = \{(7,2)_{\dot{a}_2\dot{a}_3}, (5,4)_{\dot{a}_2\bar{a}_3}\}$. Similarly, for $\bar{a}_1$ taking the union yields $\{(4,3)_{\dot{a}_2\dot{a}_3}, (2,5)_{\dot{a}_2\bar{a}_3}\ , (4,7)_{\bar{a}_2\bar{a}_3}\}$, of which the LCCS is $f^4(\bar{a}_1) = \{(4,7)_{\bar{a}_2\bar{a}_3}\}$. Adding f^4 results in the factor graph in Figure 1c.

Table 2. The two-dimensional payoff matrices for $\mathbf{u}^1(a_1, a_2)$ (left) and $\mathbf{u}^2(a_2, a_3)$ (right)

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	(4,1)	(0,0)
$\bar{a}_1$	(1,2)	(3,6)

	$\dot{a}_3$	$\bar{a}_3$
$\dot{a}_2$	(3,1)	(1,3)
$\bar{a}_2$	(0,0)	(1,1)

Finally, CMOVE eliminates agent 1. Since there are no neighboring agents left, $\mathcal{A}_i$ contains only the empty action. CMOVE takes the union of $f^4(\dot{a}_1)$ and $f^4(\bar{a}_1)$. Since $(7,2)_{\{\dot{a}_1\dot{a}_2\dot{a}_3\}}$ and $(4,7)_{\{\bar{a}_1\bar{a}_2\bar{a}_3\}}$ dominate $(5,4)_{\{\dot{a}_1\dot{a}_2\bar{a}_3\}}$, the latter is pruned, leaving $CCS = \{(7,2)_{\{\dot{a}_1\dot{a}_2\dot{a}_3\}}, (4,7)_{\{\bar{a}_1\bar{a}_2\bar{a}_3\}}\}$.

6.4 Analysis

We now analyse the correctness and complexity of CMOVE.

Theorem 2. *MOVE correctly computes the CCS.*

Proof. The proof works by induction on the number of agents. The base case is the original MO-CoG, where each $f^e(\mathbf{a}_e)$ from $\mathcal{F}$ is a singleton set. Then, since `elim` preserves the CCS (see Theorem 1), no necessary vectors are lost. When the last agent is eliminated, only one factor remains; since it is not conditioned on any agent actions and is the result of an $LCCS$ computation, it must contain one set: the CCS.

Theorem 3. *The computational complexity of CMOVE is*

$$O(\ n\ |\mathcal{A}_{max}|^{w_a}\ (w_f\ R_1 + R_2) + R_3\ ), \tag{2}$$

where w_a is the induced agent width, i.e., the maximum number of neighboring agents (connected via factors) of an agent when eliminated, w_f is the induced factor width, i.e., the maximum number of neighboring factors of an agent when eliminated, and R_1, R_2 and R_3 are the cost of applying the `prune1`, `prune2` *and* `prune3` *operators.*

Proof. CMOVE eliminates n agents and for each one computes a value (set) in a new payoff function for each joint action of the eliminated agent's neighbors. CMOVE computes $O(|\mathcal{A}_{max}|^w)$ fields per iteration, calling `prune1` for each adjacent factor, and `prune2` once after taking the union over actions of the eliminated agent. `prune3` is called only once, after eliminating all agents.

Thus, unlike nongraphical approaches, CMOVE is exponential only in the induced width, not the number of agents. In this respect, our results are similar to those for PMOVE [14]. However, those earlier complexity results do not make the effect of pruning explicit. Instead, the complexity bound makes use of problem constraints, which limit the total number of possible different value vectors. However, in practice such bounds are very loose or even impossible to define. Therefore, we instead give a description of the computational complexity that makes explicit the dependence on the effectiveness of pruning. Even though such complexity bounds are not better in the worst case (i.e., when no pruning is possible), they allow greater insight into the runtimes of the algorithms we evaluate, as is apparent in our analysis of the experimental results in Section 7.

Theorem 3 demonstrates that the complexity of CMOVE heavily depends on the runtime of its pruning operators, which in turn depends on the sizes of the input sets. The input set of prune2 is the union of what is returned by a series of applications of prune1, while prune3 uses the output of the last application of prune2. Therefore, we need to balance the effort of the lower-level pruning operators with that of the higher-level ones, which occur less often but are dependent on the output of the lower-level pruning operators. The bigger the LCCSs, the more can be gained from lower-level pruning. We compare different variants of CMOVE in the experimental section.

7 Experiments

In this section, we present an empirical analysis of CMOVE. The first goal of these experiments is to show that CMOVE, by exploiting the graphical structure to compute the CCS, can solve MO-CoGs substantially faster than both nongraphical methods and those that compute the PCS. To this end, we compare Basic CMOVE and Incremental CMOVE to the *nongraphical method* described in Section 4 and PMOVE.

We first present results on randomly generated MO-CoGs, in order to examine performance on MO-CoGs with widely varying properties. We then present results on *Mining Day*, a problem we propose as a MO-CoG benchmark, in order to establish that CMOVE performs well on a MO-CoG derived from a realistic scenario. The experiments use a C++ implementation that employs the `lp_solve` library (v5.5) to solve linear programs.

7.1 Random MO-CoGs

We employ a generation procedure for random MO-CoGs that is based on the following inputs: n, the number of agents; d, the number of payoff dimensions; ρ the number of local payoff functions; and $|\mathcal{A}_i|$, the action space size of the agents, which is the same for all agents. First, a fully connected graph with local payoff functions connected to two agents is created. Then, local payoff functions are randomly removed, while checking that the graph remains connected, until only ρ remain. The values in each local payoff function are real numbers drawn independently and uniformly from the interval $[0,10]$. All algorithms are tested on the same set of randomly generated MO-CoGs for each value of n, d, ρ, and $|\mathcal{A}_i|$ that is considered.

To compare CMOVE, PMOVE, and the nongraphical method, we tested them on random MO-CoGs with an increasing number of agents, with the average number of

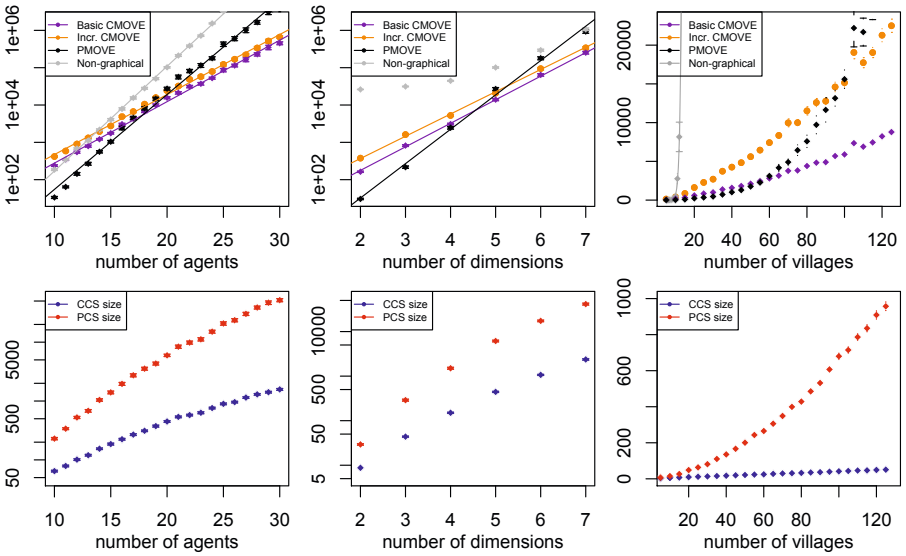

Fig. 3. Runtimes (ms) for the nongraphical method, PMOVE and CMOVE with standard errors (error bars) (top) and the corresponding number of vectors in the PCS and CCS (bottom)

factors per agent held at $\rho = 1.5n$ and the number of dimensions $d = 5$. Figure 3 (top left) shows the results, averaged over 85 MO-CoGs for each number of agents. These results demonstrate that, as the number of agents grows, using MOVE becomes key to containing the computational cost of solving the MO-CoG. CMOVE outperforms the nongraphical method from 12 agents onwards. At 25 agents, Basic CMOVE is 38 times faster. CMOVE also does significantly better than PMOVE. Though it is one order of magnitude slower with 10 agents ($238ms$ (Basic) and $416ms$ (Incremental) versus $33ms$ on average), its runtime grows much more slowly than that of PMOVE. At 20 agents, both CMOVE variants are faster than PMOVE and at 28 agents, Basic CMOVE is almost one order of magnitude faster ($228s$ versus $1{,}650s$ on average), and the difference increases with every agent.

While CMOVE's runtime grows much more slowly than that of the nongraphical method, it is still exponential in the number of agents, a counterintuitive result since the worst-case complexity is linear in the number of agents. There are two reasons for this. First, CMOVE is exponential in the induced width, which increases with the number of agents, from 3.1 at $n = 10$ to 6.0 at $n = 30$ on average, as a result of the MO-CoG generation procedure. Second, CMOVE's runtime is polynomial in the size of the CCS, and this size grows exponentially (Figure 3 (bottom left)). The fact that CMOVE is much faster than PMOVE can be explained by the sizes of the PCS and CCS, as the former grows much faster than the latter. At 10 agents, the average PCS size is 230 and the average CCS size is 65. At 30 agents, the average PCS size has risen to $51{,}745$ while the average CCS size is only $1{,}575$.

Figure 3 (top middle) compares the scalability of the algorithms in the number of objectives, on random MO-CoGs with $n = 20$ and $\rho = 30$, averaged over 100 MO-CoGs. CMOVE always outperforms the nongraphical method. Interestingly, the nongraphical

method is several orders of magnitude slower at $d = 2$, grows slowly until $d = 5$, and then starts to grow with about the same exponent as Pareto MOVE. The reason is that enumeration of all the joint actions and payoff vectors takes approximately constant time while the time it takes to prune increases exponentially. When $d = 2$, CMOVE is an order of magnitude slower than PMOVE ($163ms$ (Basic) and 377 (Incremental) versus $30ms$). However, when $d = 5$, both CMOVE variants are already faster than PMOVE and at 7 dimensions they are respectively 3.7 and 2.7 times faster. This happens because the CCS grows much more slowly than the PCS (Figure 3 (bottom middle)). The difference between Incremental and Basic CMOVE decreases as the number of dimensions increases, from a factor 2.3 at $d = 2$ to 1.3 at $d = 7$.

Overall, these results indicate that CMOVE shows large speedups over PMOVE for more than a minimal number of agents. The runtime of Incremental CMOVE grows more slowly than that of Basic CMOVE and seems favorable for large numbers of agents and high dimensions.

7.2 Mining Day

In Mining Day, a mining company mines gold and silver (objectives) from a set of mines (local payoff functions) spread throughout a geographical region (Figure 4). The mine workers live in villages also spread throughout this region. The company has one van in each village (agents) for transporting workers and must determine every morning to which mine each van should go (actions). However, vans can only travel to nearby mines (graph connectivity). Workers are more efficient if there are more workers at the mine: there is a 3% efficiency bonus per worker such that the amount of each resource mined per worker is $x \cdot 1.03^w$, where x is the base rate per worker and w is the number of workers at the mine. The base rate of gold and silver are properties of a mine. Since the company aims to maximize revenue, the best strategy depends on the prices of gold and silver, which fluctuate and are not known when the plan must be computed.

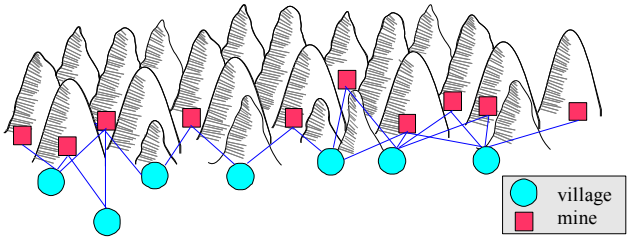

Fig. 4. The Mining Day problem

To generate a Mining Day instance with v villages (agents), we randomly assign 2-5 workers to each village and connect it to 2-4 mines. Each village is only connected to mines with a greater or equal index, i.e., if village i is connected to m mines, it is connected to mines i to $i + m - 1$. The last village is connected to 4 mines and thus the number of mines is $v + 3$. The base rates per worker for each resource at each mine are drawn uniformly and independently from [0,10].

The results for the mining day problem are shown in Figure 3 (top right). The runtime of the nongraphical method grows exponentially with the number of agents. At only 13

agents, the runtime is already more than $30s$. By contrast, both CMOVE and PMOVE are able to tackle problems with over 100 agents within that timeframe. In addition, the runtime of PMOVE grows much more quickly than that of CMOVE. In this two-dimensional setting, Basic CMOVE is better than Incremental CMOVE. Basic CMOVE and PMOVE both have runtimes of around $2.8s$ at 60 agents, but at 100 agents, Basic CMOVE runs in about $5.9s$ and PMOVE in $21s$. Even though Incremental CMOVE is worse than Basic CMOVE, its runtime still grows a lot slower than PMOVE, and beats PMOVE when there are many agents.

The difference between PMOVE and CMOVE results from the relationship between the number of agents and the sizes of the CCS, which grows linearly, and the PCS, which grows polynomially (Figure 3 (bottom right)). The induced width remains around 4 regardless of v. These results demonstrate that, when the CS grows linearly (or polynomially) in the number of agents, MOVE can solve MO-CoGs with many more agents than the nongraphical approach. In problems where the CCS grows more slowly than the PCS, CMOVE can solve MO-CoGs with many more agents than PMOVE.

8 Conclusions and Future Work

In this paper, we proposed the CMOVE algorithm for multi-objective coordination graphs. Unlike previous methods, it computes the convex coverage set (CCS) rather than the Pareto coverage set (PCS). Not only does this provide the optimal solution when the scalarization function is linear or stochastic strategies are allowed, it also greatly reduces computational costs.

Using two variants of CMOVE – based on the trade-off between pruning effort and and smaller intermediate results – we analyzed CMOVE's complexity in terms of the different pruning operators that can be used to compute the local CCSs. Our empirical study showed that CMOVE can tackle multi-objective problems much faster than methods that compute the PCS. The runtime of CMOVE grows much more slowly than that of PMOVE because the CCS grows much more slowly than the PCS. Therefore, we conclude that computing the CCS is key to keeping large MO-CoGs tractable.

In future work, we hope to develop approximate techniques for MO-CoGs. The work of [5], which converts graphs to trees and applies *max-plus* [9] to approximate the PCS, could be extended to approximate the CCS. Alternatively, an efficient multi-objective version of max-plus for graphs with loops could also approximate the CCS. In addition, loosening the definition of the CCS, in the spirit of the ϵ-approximate Pareto front [3], could also yield efficient approximations. Finally, we hope to develop a multi-objective version of *sparse cooperative Q-learning* [9] that would use CMOVE as a subroutine to tackle sequential multi-objective multi-agent tasks.

Acknowledgements. This research is supported by the NWO DTC-NCAP (#612.001.109) and NWO CATCH (#640.005.003) projects.

References

1. Barrett, L., Narayanan, S.: Learning all optimal policies with multiple criteria. In: ICML, pp. 41–47. ACM, New York (2008)

2. Becker, R., Zilberstein, S., Lesser, V., Goldman, C.V.: Transition-Independent Decentralized Markov Decision Processes. In: AAMAS (2003)
3. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. CoRR, abs/1104.3489 (2011)
4. Cassandra, A.R., Littman, M.L., Zhang, N.L.: Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In: UAI, pp. 54–61 (1997)
5. Delle Fave, F.M., Stranders, R., Rogers, A., Jennings, N.R.: Bounded decentralised coordination over multiple objectives. In: AAMAS, pp. 371–378 (2011)
6. Dubus, J.-P., Gonzales, C., Perny, P.: Choquet optimization using gai networks for multi-agent/multicriteria decision-making. In: Rossi, F., Tsoukias, A. (eds.) ADT 2009. LNCS, vol. 5783, pp. 377–389. Springer, Heidelberg (2009)
7. Feng, Z., Zilberstein, S.: Region-based incremental pruning for POMDPs. CoRR, abs/1207.4116 (2012)
8. Guestrin, C.E., Koller, D., Parr, R.: Multiagent planning with factored MDPs. In: NIPS (2002)
9. Kok, J.R., Vlassis, N.: Collaborative multiagent reinforcement learning by payoff propagation. J. Mach. Learn. Res. 7, 1789–1828 (2006)
10. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)
11. Lizotte, D.J., Bowling, M., Murphy, S.A.: Efficient reinforcement learning with multiple reward functions for randomized clinical trial analysis. In: ICML, pp. 695–702 (2010)
12. Marinescu, R., Razak, A., Wilson, N.: Multi-objective influence diagrams. In: UAI (2012)
13. Roijers, D.M., Whiteson, S., Oliehoek, F.A.: Multi-objective variable elimination for collaborative graphical games. In: AAMAS (2013) (Extended Abstract)
14. Rollón, E.: Multi-Objective Optimization for Graphical Models. PhD thesis, Universitat Politècnica de Catalunya (2008)
15. Rollón, E., Larrosa, J.: Bucket elimination for multiobjective optimization problems. Journal of Heuristics 12, 307–328 (2006)
16. Tesauro, G., Das, R., Chan, H., Kephart, J.O., Lefurgy, C., Levine, D.W., Rawson, F.: Managing power consumption and performance of computing systems using reinforcement learning. In: NIPS (2007)
17. Vamplew, P., Dazeley, R., Barker, E., Kelarev, A.: Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In: Nicholson, A., Li, X. (eds.) AI 2009. LNCS, vol. 5866, pp. 340–349. Springer, Heidelberg (2009)