Journal of Artificial Intelligence Research 52 (2015) 399–443 Submitted 08/14; published 03/15

Computing Convex Coverage Sets for Faster Multi-objective Coordination

Diederik M. Roijers D.M.ROIJERS@UVA.NL
Shimon Whiteson S.A.WHITESON@UVA.NL
Frans A. Oliehoek F.A.OLIEHOEK@UVA.NL
Informatics Institute
University of Amsterdam
Amsterdam, The Netherlands

Abstract

In this article, we propose new algorithms for *multi-objective coordination graphs (MO-CoGs)*. Key to the efficiency of these algorithms is that they compute a *convex coverage set (CCS)* instead of a *Pareto coverage set (PCS)*. Not only is a CCS a sufficient solution set for a large class of problems, it also has important characteristics that facilitate more efficient solutions. We propose two main algorithms for computing a CCS in MO-CoGs. *Convex multi-objective variable elimination (CMOVE)* computes a CCS by performing a series of agent eliminations, which can be seen as solving a series of local multi-objective subproblems. *Variable elimination linear support (VELS)* iteratively identifies the single weight vector $\mathbf{w}$ that can lead to the maximal possible improvement on a partial CCS and calls *variable elimination* to solve a scalarized instance of the problem for $\mathbf{w}$. VELS is faster than CMOVE for small and medium numbers of objectives and can compute an ε-approximate CCS in a fraction of the runtime. In addition, we propose variants of these methods that employ AND/OR tree search instead of variable elimination to achieve memory efficiency. We analyze the runtime and space complexities of these methods, prove their correctness, and compare them empirically against a naive baseline and an existing PCS method, both in terms of memory-usage and runtime. Our results show that, by focusing on the CCS, these methods achieve much better scalability in the number of agents than the current state of the art.

1. Introduction

In many real-world problem domains, such as maintenance planning (Scharpff, Spaan, Volker, & De Weerdt, 2013) and traffic light control (Pham et al., 2013), multiple agents need to coordinate their actions in order to maximize a common utility. Key to coordinating efficiently in these domains is exploiting *loose couplings* between agents (Guestrin, Koller, & Parr, 2002; Kok & Vlassis, 2004): each agent's actions directly affect only a subset of the other agents.

Multi-agent coordination is complicated by the fact that, in many domains, agents need to balance multiple objectives (Roijers, Vamplew, Whiteson, & Dazeley, 2013a). For example, agents might have to maximize the performance of a computer network while minimizing power consumption (Tesauro, Das, Chan, Kephart, Lefurgy, Levine, & Rawson, 2007), or maximize the cost efficiency of maintenance tasks on a road network while minimizing traffic delays (Roijers, Scharpff, Spaan, Oliehoek, de Weerdt, & Whiteson, 2014).

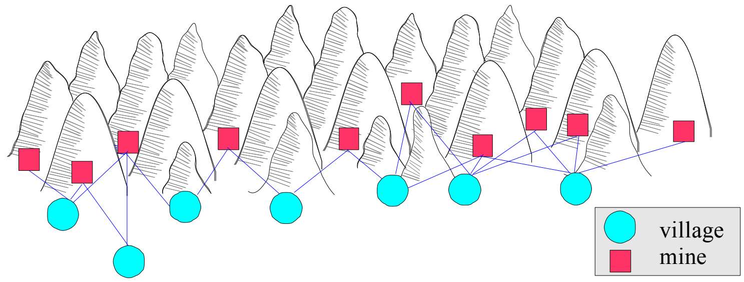


Figure 1: Mining company example.

However, the presence of multiple objectives does not per se necessitate the use of specialized multi-objective solution methods. If the problem can be *scalarized*, i.e., the utility function can be converted to a scalar utility function, the problem may be solvable with existing single-objective methods. Such a conversion involves two steps (Roijers et al., 2013a). The first step is to specify a *scalarization function*.

Definition 1. *A* scalarization function f*, is a function that maps a multi-objective utility of a solution* $\mathbf{a}$ *of a decision problem,* $\mathbf{u}(\mathbf{a})$*, to a scalar utility* $u_{\mathbf{w}}(\mathbf{a})$*:*

$$u_{\mathbf{w}}(\mathbf{a}) = f(\mathbf{u}(\mathbf{a}), \mathbf{w}),$$

where $\mathbf{w}$ *is a weight vector that parameterizes* f*.*

The second step is to define a single-objective version of the decision problem such that the utility of each solution $\mathbf{a}$ equals the scalarized utility of the original problem $u_{\mathbf{w}}(\mathbf{a})$.

Unfortunately, scalarizing the problem before solving it is not always possible because $\mathbf{w}$ may not be known in advance. For example, consider a company that mines different resources. In Figure 1, we depict the problem this company faces: in the morning one van per village needs to transport workers from that village to a nearby mine, where various resources will be mined. Different mines yield different quantities of resource per worker. The market prices per resource vary through a stochastic process and every price change can alter the optimal assignment of vans. The expected price variation increases with the passage of time. To maximize performance, it is thus critical to act based on the latest possible price information. Since computing the optimal van assignment takes time, redoing this computation for every price change is highly undesirable.

In such settings, we need a multi-objective method that computes, in advance, an optimal solution for all possible prices, $\mathbf{w}$. We call such a set a *coverage set (CS)*. In many cases, $\mathbf{w}$ is revealed before a solution must be executed, in which case that solution can be automatically selected from the CS given $\mathbf{w}$. In other cases, $\mathbf{w}$ is never made explicit but instead a human is involved in the decision making and selects one solution from the CS, perhaps on the basis of constraints or preferences that were too difficult to formalize in the objectives themselves (Roijers et al., 2013a). In both cases, because the CS is typically much smaller than the complete set of solutions, selecting the optimal joint action from the CS is typically much easier than selecting it directly from the complete set of solutions.

In this article, we consider how multi-objective methods can be made efficient for problems that require the coordination of multiple, loosely coupled agents. In particular, we address *multi-objective coordination graphs (MO-CoGs)*: one-shot multi-agent decision problems in which loose couplings are expressed using a graphical model. MO-CoGs form an important class of decision problems. Not only can they be used to model a variety of real-world problems (Delle Fave, Stranders, Rogers, & Jennings, 2011; Marinescu, 2011; Rollón, 2008), but many sequential decision problems can be modeled as a series of MO-CoGs, as is common in single-objective problems (Guestrin et al., 2002; Kok & Vlassis, 2004; Oliehoek, Spaan, Dibangoye, & Amato, 2010).

Key to the efficiency of the MO-CoG methods we propose is that they compute a *convex coverage set (CCS)* instead of a *Pareto coverage set (PCS)*. The CCS is a subset of the PCS that is a sufficient solution for any multi-objective problem with a linear scalarization function. For example, in the mining company example of Figure 1, f is linear, since the total revenue is simply the sum of the quantity of each resource mined times its price per unit. However, even if f is nonlinear, if stochastic solutions are allowed, then a CCS is again sufficient.[1]

The CCS has not previously been considered as a solution concept for MO-CoGs because computing a CCS requires running linear programs, whilst computing a PCS requires only pairwise comparisons of solutions. However, a key insight of this article[2] is that, in loosely coupled systems, CCSs are easier to compute than PCSs, for two reasons. First, the CCS is a (typically much smaller) subset of the PCS. In loosely coupled settings, efficient methods work by solving a series of local subproblems; focusing on the CCS can greatly reduce the size of these subproblems. Second, focusing on the CCS makes solving a MO-CoG equivalent to finding an optimal *piecewise-linear and convex (PWLC)* scalarized value function, for which efficient techniques can be adapted. For these reasons, we argue that the CCS is often the concept of choice for MO-CoGs.

We propose two approaches that exploit these insights to solve MO-CoGs more efficiently than existing methods (Delle Fave et al., 2011; Dubus, Gonzales, & Perny, 2009; Marinescu, Razak, & Wilson, 2012; Rollón & Larrosa, 2006). The first approach deals with the multiple objectives on the level of individual agents, while the second deals with them on a global level.

The first approach extends an algorithm by Rollón and Larrosa (2006) which we refer to as *Pareto multi-objective variable elimination (PMOVE)*[3], that computes local Pareto sets at each agent elimination, to compute a CCS instead. We call the resulting algorithm *convex multi-objective variable elimination (CMOVE)*.

The second approach is a new abstract algorithm that we call *optimistic linear support (OLS)* and is much faster for small and medium numbers of objectives. Furthermore, OLS

1. To be precise, in the case of stochastic strategies a CCS of deterministic strategies is always sufficient (Vamplew, Dazeley, Barker, & Kelarev, 2009); in the case of deterministic strategies, linearity of the scalarization function makes the CCS sufficient (Roijers et al., 2013a).
2. This article synthesizes and extends research already reported in two conference papers. Specifically, the CMOVE algorithm (Section 4) was previously published at ADT (Roijers, Whiteson, & Oliehoek, 2013b) and the VELS algorithm (Section 5) at AAMAS (Roijers, Whiteson, & Oliehoek, 2014). The memory-efficient methods for computing CCSs (Section 6) are a novel contribution of this article.
3. In the original article, this algorithm is called *multi-objective bucket elimination (MOBE)*. However, we use PMOVE to be consistent with the names of the other algorithms mentioned in this article.

can be used to produce a bounded approximation of the CCS, an ε-CCS, if there is not enough time to compute a full CCS. OLS is a generic method that employs single-objective solvers as a subroutine. In this article, we consider two implementations of this subroutine. Using *variable elimination (VE)* as a subroutine yields *variable elimination linear support (VELS)*, which is particularly fast for small and moderate numbers of objectives and is more memory-efficient than CMOVE. However, when memory is highly limited, this reduction in memory usage may not be enough. In such cases, using AND/OR search (Mateescu & Dechter, 2005) instead of VE yields *AND/OR tree search linear support (TSLS)*, which is slower than VELS but much more memory efficient.

We prove the correctness of both CMOVE and OLS. We analyze the runtime and space complexities of both methods and show that our methods have better guarantees than PCS methods. We show CMOVE and OLS are complementary, i.e., various trade-offs exist between them and their variants.

Furthermore, we demonstrate empirically, on both randomized and more realistic problems, that CMOVE and VELS scale much better than previous algorithms. We also empirically confirm the trade-offs between CMOVE and OLS. We show that OLS, when used as a bounded approximation algorithm, can save additional orders of magnitude of runtime, even for small ε. Finally, we show that, even when memory is highly limited, TSLS can still solve large problems.

The rest of this article is structured as follows. First, we provide a formal definition of our model, as well as an overview of existing solution methods in Section 2. After presenting a naive approach in Section 3, in Sections 4, 5 and 6, we analyze the runtime and space complexities of each algorithm, and compare them empirically, against each other and existing algorithms, at the end of each section. Finally, we conclude in Section 7 with an overview of our contributions and findings, and suggestions for future research.

2. Background

In this section, we formalize the *multi-objective coordination graph (MO-CoG)*. Before doing so however, we describe the single-objective version of this problem, the *coordination graph* (CoG), of which the MO-CoG is an extension, and the *variable elimination (VE)* algorithm for solving CoGs. The methods we present in Section 4 and 5 build on VE in different ways.

2.1 (Single-Objective) Coordination Graphs

A *coordination graph* (CoG) (Guestrin et al., 2002; Kok & Vlassis, 2004) is a tuple $\langle \mathcal{D}, \mathcal{A}, \mathcal{U} \rangle$, where

- $\mathcal{D} = \{1, ..., n\}$ 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. A joint action is thus a tuple containing an action for each agent $\mathbf{a} = \langle a_1, ..., a_n \rangle$, and
- $\mathcal{U} = \left\{ u^1, ..., u^\rho \right\}$ is the set of ρ scalar *local payoff functions*, each of which has limited *scope*, i.e., it depends on only a subset of the agents. The total team payoff is the sum of the local payoffs: $u(\mathbf{a}) = \sum_{e=1}^{\rho} u^e(\mathbf{a}_e)$.

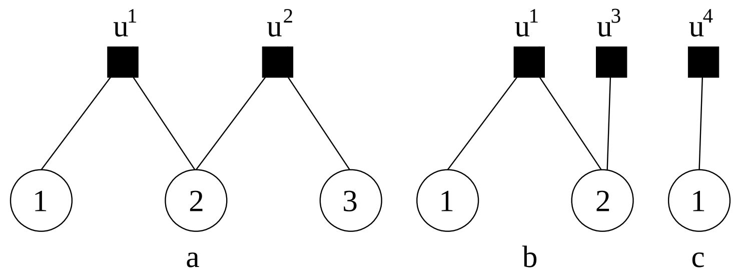


Figure 2: (a) A CoG with 3 agents and 2 local payoff functions (b) after eliminating agent 3 by adding u^3 (c) after eliminating agent 2 by adding u^4.

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	3.25	0
$\bar{a}_1$	1.25	3.75

	$\dot{a}_3$	$\bar{a}_3$
$\dot{a}_2$	2.5	1.5
$\bar{a}_2$	0	1

Table 1: The payoff matrices for $u^1(a_1, a_2)$ (left) and $u^2(a_2, a_3)$ (right). There are two possible actions per agent, denoted by a dot ($\dot{a}_1$) and a bar ($\bar{a}_1$).

All agents share the payoff function $u(\mathbf{a})$. We abuse the notation e to both index a local payoff function u^e and to denote the subset of agents in its scope; $\mathbf{a}_e$ is thus a *local joint action*, i.e., a joint action of this subset of agents.

The decomposition of $u(\mathbf{a})$ into local payoff functions can be represented as a *factor graph* (Bishop, 2006), a bipartite graph containing two types of vertices: agents (variables) and local payoff functions (factors), with edges connecting local payoff functions to the agents in their scope.

Figure 2a shows the factor graph of an example CoG in which the team payoff function decomposes into two local payoff functions, each with two agents in scope:

$$u(\mathbf{a}) = \sum_{e=1}^{\rho} u^e(\mathbf{a}_e) = u^1(a_1, a_2) + u^2(a_2, a_3).$$

The local payoff functions are defined in Table 1. The factor graph illustrates the loose couplings that result from the decomposition into local payoff functions. In particular, each agent's choice of action directly depends only on those of its immediate neighbors, e.g., once agent 1 knows agent 2's action, it can choose its own action without considering agent 3.

2.2 Variable Elimination

We now discuss the *variable elimination (VE)* algorithm, on which several multi-objective extensions (Rollón & Larrosa, 2006; Rollón, 2008) build, including our own CMOVE algorithm (Section 4). We also use VE as a subroutine in the OLS algorithm (Section 5).

VE exploits the loose couplings expressed by the local payoff functions to efficiently compute the optimal joint action, i.e., the joint action maximizing $u(\mathbf{a})$. First, in the *forward* *pass*, VE eliminates each of the agents in turn by computing the value of that agent's best response to every possible joint action of its neighbors. These values are used to construct a new local payoff function that encodes the value of the best response and replaces the agent and the payoff functions in which it participated. In the original algorithm, once all agents are eliminated, a *backward pass* assembles the optimal joint action using the constructed payoff functions. Here, we present a slight variant in which each payoff is 'tagged' with the action that generates it, obviating the need for a backwards pass. While the two algorithms are equivalent, this variant is more amenable to the multi-objective extension we present in Section 4.

VE eliminates agents from the graph in a predetermined order. Algorithm 1 shows pseudocode for the elimination of a single agent i. First, VE determines the set of local payoff functions connected to i, $\mathcal{U}_i$, and the neighboring agents of i, n_i (lines 1-2).

Definition 2. *The set of neighboring local payoff functions $\mathcal{U}_i$ of i is the set of all local payoff functions that have agent i in scope.*

Definition 3. *The set of neighboring agents of i, n_i, is the set of all agents that are in scope of one or more of the local payoff functions in $\mathcal{U}_i$.*

Then, it constructs a new payoff function by computing the value of agent i's best response to each possible joint action $\mathbf{a}_{n_i}$ of the agents in n_i (lines 3-12). To do so, it loops over all these joint actions $\mathcal{A}_{n_i}$ (line 4). For each $\mathbf{a}_{n_i}$, it loops over all the actions $\mathcal{A}_i$ available to agent i (line 6). For each $a_i \in \mathcal{A}_i$, it computes the local payoff when agent i responds to $\mathbf{a}_{n_i}$ with a_i (line 7). VE tags the total payoff with a_i, the action that generates it (line 8) in order to be able to retrieve the optimal joint action later. If there are already tags present, VE appends a_i to them; in this way, the entire joint action is incrementally constructed. VE maintains the value of the best response by taking the maximum of these payoffs (line 11). Finally, it eliminates the agent and all payoff functions in $\mathcal{U}_i$ and replaces them with the newly constructed local payoff function (line 13).

Algorithm 1: `elimVE`$(\mathcal{U}, i)$

Input: A CoG $\mathcal{U}$, and an agent i
1 $\mathcal{U}_i \leftarrow$ set of local payoff functions involving i
2 $n_i \leftarrow$ set of neighboring agents of i
3 $u^{new} \leftarrow$ a new factor taking joint actions of n_i, $\mathbf{a}_{n_i}$, as input
4 **foreach** $\mathbf{a}_{n_i} \in \mathcal{A}_{n_i}$ **do**
5     $\mathcal{S} \leftarrow \emptyset$
6   **foreach** $a_i \in \mathcal{A}_i$ **do**
7         $v \leftarrow \sum_{u_j \in \mathcal{U}_i} u_j(\mathbf{a}_{n_i}, a_i)$
8   tag v with a_i
9         $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
10   **end**
11     $u^{new}(\mathbf{a}_{n_i}) \leftarrow \max(\mathcal{S})$
12 **end**
13 **return** $(\mathcal{U} \setminus \mathcal{U}_i) \cup \{u^{new}\}$

Consider the example in Figure 2a and Table 1. The optimal payoff maximizes the sum of the two payoff functions:

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_1,a_2,a_3} u^1(a_1, a_2) + u^2(a_2, a_3).$$

If VE eliminates agent 3 first, then it pushes the maximization over a_3 inward such that goes only over the local payoff functions involving agent 3, in this case just u^2:

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_1,a_2} \left(u^1(a_1, a_2) + \max_{a_3} u^2(a_2, a_3) \right).$$

VE solves the inner maximization and replaces it with a new local payoff function u^3 that depends only on agent 3's neighbors, thereby eliminating agent 1:

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_1,a_2} \left(u^1(a_1, a_2) + u^3(a_2) \right),$$

which leads to the new factor graph depicted in Figure 2b. The values of $u^3(a_2)$ are $u^3(\dot{a}_2) = 2.5$, using $\dot{a}_3$, and $u^3(\bar{a}_2) = 1$ using $\bar{a}_3$, as these are the optimal payoffs for the actions of agent 2, given the payoffs shown in Table 1. Because we ultimately want the optimal joint action, not just the optimal payoff, VE tags each payoff of u^3 with the action of agent 3 that generates it, i.e., we can think of $u^3(a_2)$ as a $(\texttt{value}, \texttt{tag})$ pair. We denote such a pair with parentheses and a subscript: $u^3(\dot{a}_2) = (2.5)_{\dot{a}_3}$, and $u^3(\bar{a}_2) = (1)_{\bar{a}_3}$.

VE next eliminates agent 2, yielding the factor graph shown in Figure 2c:

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_1} \left(\max_{a_2} u^1(a_1, a_2) + u^3(a_2) \right) = \max_{a_1} u^4(a_1).$$

VE appends the new tags for agent 2 to the existing tags for agent 3, yielding the following tagged payoff values: $u^4(\dot{a}_1) = \max_{a_2} u^1(\dot{a}_1, a_2) + u^3(a_2) = (3.25)_{\dot{a}_2} + (2.5)_{\dot{a}_2\dot{a}_3} = (5.75)_{\dot{a}_2\dot{a}_3}$ and $u^4(\bar{a}_1) = (3.75)_{\bar{a}_2} + (1)_{\bar{a}_2\bar{a}_3} = (4.75)_{\bar{a}_2\bar{a}_3}$. Finally, maximizing over a_1 yields the optimal payoff of $(5.75)_{\dot{a}_1\dot{a}_2\dot{a}_3}$, with the optimal action contained in the tags.

The runtime complexity of VE is exponential, not in the number of agents, but only in the *induced width*, which is often much less than the number of agents.

Theorem 1. *The computational complexity of VE is $O(n|\mathcal{A}_{max}|^w)$ where $|\mathcal{A}_{max}|$ is the maximal number of actions for a* single agent *and w is the induced width, i.e., the maximal number of neighboring agents of an agent plus one (the agent itself), at the moment when it is eliminated (Guestrin et al., 2002).*

Theorem 2. *The space complexity of VE is $O(\ n\ |\mathcal{A}_{max}|^w)$.*

This space complexity arises because, for every agent elimination, a new local payoff function is created with $O(|\mathcal{A}_{max}|^w)$ fields (possible input actions). Since it is impossible to tell a priori how many of these new local payoff functions exist at any given time during the execution of VE, this need to be multiplied by the total number of new local payoff functions created during a VE execution, which is n.

While VE is designed to minimize runtime[4] other methods focus on memory efficiency instead (Mateescu & Dechter, 2005). We discuss memory efficiency further in Section 6.1.

4. In fact, VE is proven to have the best runtime guarantees within a large class of algorithms (Rosenthal, 1977).

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	(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)

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).

2.3 Multi-objective Coordination Graphs

A *multi-objective coordination graph* (MO-CoG) is a tuple $\langle \mathcal{D}, \mathcal{A}, \mathcal{U} \rangle$ in which $\mathcal{D}$ and $\mathcal{A}$ are as before but, $\mathcal{U} = \{\mathbf{u}^1, ..., \mathbf{u}^\rho\}$ is now a set of ρ, d-dimensional *local payoff functions.* The total team payoff is the sum of local vector-valued payoffs: $\mathbf{u}(\mathbf{a}) = \sum_{e=1}^{\rho} \mathbf{u}^e(\mathbf{a}_e)$. We use u_i to indicate the value of the i-th objective. We denote the set of all possible joint action values as $\mathcal{V}$. Table 2 shows a two-dimensional MO-CoG with the same structure as the single-objective example in Section 2.1, but with multi-objective payoffs.

The solution to a MO-CoG is a *coverage set (CS)* of joint actions $\mathbf{a}$ and associated values $\mathbf{u}(\mathbf{a})$ that contains at least one optimal joint action for each possible parameter vector $\mathbf{w}$ of the scalarization function f (Definition 1). A CS is a subset of the *undominated set*:

Definition 4. *The* undominated set (U) *of a MO-CoG, is the set of all joint actions and associated payoff values that are optimal for some* $\mathbf{w}$ *of the scalarization function* f*.*

$$U(\mathcal{V}) = \left\{\mathbf{u}(\mathbf{a}) : \ \mathbf{u}(\mathbf{a}) \in \mathcal{V} \ \wedge \ \exists \mathbf{w} \forall \mathbf{a}' \ u_{\mathbf{w}}(\mathbf{a}) \geq u_{\mathbf{w}}(\mathbf{a}')\right\}.$$

Because we care about having at least one optimal joint action for every $\mathbf{w}$, rather than all optimal joint actions, a *lossless* subset of U suffices:

Definition 5. *A* coverage set (CS)*,* $CS(\mathcal{V})$*, is a subset of* U*, such that for each possible* $\mathbf{w}$*, there is at least one optimal solution in the CS, i.e.,*

$$\forall \mathbf{w} \exists \mathbf{a} \quad \left(\mathbf{u}(\mathbf{a}) \in CS(\mathcal{V}) \wedge \forall \mathbf{a}' \ u_{\mathbf{w}}(\mathbf{a}) \geq u_{\mathbf{w}}(\mathbf{a}')\right).$$

Note that the CS is not necessarily unique. Typically we seek the smallest possible CS. For convenience, we assume that payoff vectors in the CS contain both the values and associated joint actions, as suggested by the tagging scheme described in Section 2.2.

Which payoff vectors from $\mathcal{V}$ should be in the CS depends on what we know about the scalarization function f. A minimal assumption is that f is monotonically increasing, i.e., if the value for one objective u_i, increases while all $u_{j \neq i}$ stay constant, the scalarized value $\mathbf{u}(\mathbf{a})$ cannot decrease. This assumption ensures that objectives are desirable, i.e., all else being equal, having more of them is always better.

Definition 6. *The* Pareto front *is the undominated set for arbitrary strictly monotonically increasing scalarization functions* f*.*

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

where $\succ_P$ *indicates* Pareto dominance (P-dominance)*: greater or equal in all objectives and strictly greater in at least one objective.*

In order to have all optimal scalarized values, it is not necessary to compute the entire PF. E.g., if two joint actions have equal payoffs we need to retain only one of those.

Definition 7. *A* Pareto coverage set (PCS)*, $PCS(\mathcal{V}) \subseteq PF(\mathcal{V})$, is a coverage set for arbitrary strictly monotonically increasing scalarization functions f, i.e.,*

$$\forall \mathbf{a}' \exists \mathbf{a} \quad \big(\mathbf{u}(\mathbf{a}) \in PCS(\mathcal{V}) \wedge (\mathbf{u}(\mathbf{a}) \succ_P \mathbf{u}(\mathbf{a}') \vee \mathbf{u}(\mathbf{a}) = \mathbf{u}(\mathbf{a}'))\ \big).$$

Computing P-dominance requires only pairwise comparison of payoff vectors (Feng & Zilberstein, 2004).[5]

A highly prevalent scenario is that, in addition to f being monotonically increasing, we also know that it is linear, that is, the parameter vectors $\mathbf{w}$ are weights by which the values of the individual objectives are multiplied, $f = \mathbf{w} \cdot \mathbf{u}(\mathbf{a})$. In the mining example from Figure 1, resources are traded on an open market and all resources have a positive unit price. In this case, the scalarization is a linear combination of the amount of each resource mined, where the weights correspond to the price per unit of each resource. Many more examples of linear scalarization functions exist in the literature, e.g., (Lizotte, Bowling, & Murphy, 2010). 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. In this case, only a *convex coverage set (CCS)* is needed, which is a subset of the *convex hull (CH)*[6]:

Definition 8. *The* convex hull (CH) *is the undominated set for linear non-decreasing scalarizations $f(\mathbf{u}(\mathbf{a}), \mathbf{w}) = \mathbf{w} \cdot \mathbf{u}(\mathbf{a})$:*

$$CH(\mathcal{V}) = \big\{\mathbf{u}(\mathbf{a}) : \mathbf{u}(\mathbf{a}) \in \mathcal{V} \ \wedge \ \exists \mathbf{w} \forall \mathbf{a}' \ \mathbf{w} \cdot \mathbf{u}(\mathbf{a}) \geq \mathbf{w} \cdot \mathbf{u}(\mathbf{a}')\big\}.$$

That is, the CH contains all solutions that attain the optimal value for at least one weight. Vectors not in the CH are *C-dominated*. In contrast to P-domination, C-domination cannot be tested for with pairwise comparisons because it can take two or more payoff vectors to C-dominate a payoff vector. Note that the CH contains more solutions than needed to guarantee an optimal scalarized value value: it can contain multiple solutions that are optimal for one specific weight. A lossless subset of the CH with respect to linear scalarizations is called a *convex coverage set (CCS)*, i.e., a CCS retains at least one $\mathbf{u}(\mathbf{a})$ that maximizes the scalarized payoff, $\mathbf{w} \cdot \mathbf{u}(\mathbf{a})$, for every $\mathbf{w}$:

Definition 9. *A* convex coverage set (CCS)*, $CCS(\mathcal{V}) \subseteq CH(\mathcal{V})$, is a CS for linear non-decreasing scalarizations, i.e.,*

$$\forall \mathbf{w} \exists \mathbf{a} \quad \big(\mathbf{u}(\mathbf{a}) \in CCS(\mathcal{V}) \wedge \forall \mathbf{a}' \ \mathbf{w} \cdot \mathbf{u}(\mathbf{a}) \geq \mathbf{w} \cdot \mathbf{u}(\mathbf{a}')\big).$$

Since linear non-decreasing functions are a specific type of monotonically increasing function, there is always a CCS that is a subset of the smallest possible PCS.

As previously mentioned, CSs like the PCS and CCS, may not be unique. For example, if there are two joint actions with equal payoff vectors, we need at most one of them to make a PCS or CCS.

5. P-dominance is often called *pairwise dominance* in the POMDP literature.
6. Note that the term *convex hull* is overloaded. In graphics, the convex hull is a superset of what we mean by the convex hull in this article.

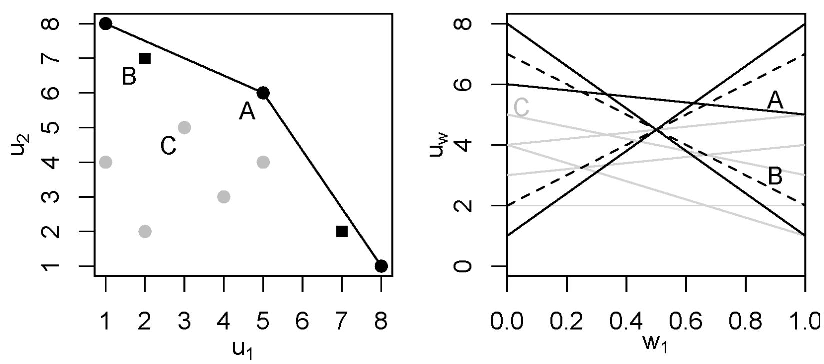


Figure 3: The CCS (filled circles at left, and solid black lines at right) versus the PCS (filled circles and squares at left, and both dashed and solid black lines at right) for twelve random 2-dimensional payoff vectors.

In practice, the PCS and the CCS are often equal to the PF and CH. However, the algorithms proposed in this article are guaranteed to produce a PCS or a CCS, and not necessarily the entire PF or the CH. Because PCSs and the CCSs are sufficient solutions in terms of scalarized value, we say that these algorithms solve the MO-CoGs.

In Figure 3 (left) the values of joint actions, $\mathbf{u}(\mathbf{a})$, are represented as points in value-space, for a two-objective MO-CoG. The joint action value 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 3 (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.

Many multi-objective methods, e.g., (Delle Fave et al., 2011; Dubus et al., 2009; Marinescu et al., 2012; Rollón, 2008) simply assume that the PCS is the appropriate solution concept. However, we argue that the choice of CS depends on 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 the scalarization function is linear, and that in such cases one should use the CCS.

In addition to the shape of f, the choice of solution concept depends on whether only deterministic joint actions are considered or whether stochastic strategies are also permitted. A stochastic strategy π assigns a probability to each joint action $\mathcal{A} \to [0, 1]$. The probabilities for all joint actions together sum to 1, $\sum_{\mathbf{a} \in \mathcal{A}} \pi(\mathbf{a}) = 1$. The value of a stochastic strategy is a linear combination of the value vectors of the joint actions of which it is a mixture: $\mathbf{u}^\pi = \sum_{\mathbf{a} \in \mathcal{A}} \pi(\mathbf{a})\mathbf{u}(\mathbf{a})$. Therefore, the optimal values, for any monotonically increasing f, lie on the convex upper surface spanned by the strategies in the CCS, as indicated by the lines in Figure 3 (left). Therefore, all optimal values for monotonically increasing f, including nonlinear ones, can be constructed by taking mixture policies from the CCS (Vamplew et al., 2009).

This article considers methods for computing CCSs, which, as we show in Sections 4 and 5, can be computed more efficiently than PCSs. Furthermore, CCSs are typically much smaller. This is particularly important when the final selection of the joint is done by (a group of) humans, who have to compare all possible alternatives in the solution set.

The methods presented in this article are based on *variable elimination (VE)* (Sections 4 and 5) and *AND/OR tree search (TS)* (Section 6). These algorithms are exact solution methods for CoGs.

The CMOVE algorithm we propose in Section 5 is based on VE. It differs from another multi-objective algorithm based on VE, which we refer to as PMOVE (Rollón & Larrosa, 2006), in that it produces a CCS rather than a PCS. An alternative to VE are message-passing algorithms, like *max-plus* (Pearl, 1988; Kok & Vlassis, 2006a). However, these are guaranteed to be exact only for tree-structured CoGs. Multi-objective methods that build on max-plus such as that of Delle Fave et al. (2011), have this same limitation, unless they preprocess the CoG to form a clique-tree or GAI network (Dubus et al., 2009). On tree structured graphs, both message-passing algorithms and VE produce optimal solutions with similar runtime guarantees. Note that, like PMOVE, existing multi-objective methods based on message passing produce a PCS rather than a CCS.

In Section 5, we take a different approach to multi-objective coordination based on an *outer loop approach*. As we explain, this approach is applicable only for computing a CCS, not a PCS, but has considerable advantages in terms of runtime and memory usage.

3. Non-graphical Approach

A naive way to compute a 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. 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. The initial 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 $\mathbf{a}$:

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

where 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:

$$A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A \wedge \mathbf{b} \in B\}.$$

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 $\mathcal{V}$:

$$CCS(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\bigcup_{\mathbf{a}} \bigoplus_{f^e \in \mathcal{F}} f^e(\mathbf{a}_e)). \tag{1}$$

The non-graphical CCS algorithm simply computes the righthand side of Equation 1, i.e., it computes $\mathcal{V}(\mathcal{F})$ explicitly by looping over all actions, and for each action looping over all local VSFs, and then pruning that set down to a CCS.

A CCS contains at least one payoff vector that maximizes the scalarized value for every $\mathbf{w}$:

$$\forall \mathbf{w} \quad \Big(\mathbf{a} = \arg\max_{\mathbf{a} \in \mathcal{A}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a})\Big) \implies \exists \mathbf{a}' \quad \mathbf{u}(\mathbf{a}') \in CCS(\mathcal{V}(\mathcal{F})) \ \wedge \ \mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \mathbf{w} \cdot \mathbf{u}(\mathbf{a}'). \quad (2)$$

That is, for every $\mathbf{w}$ there is an solution $\mathbf{a}'$ that is part of the CCS and that achieves the same value as a maximizing solution $\mathbf{a}$. Moreover the value of such solutions is given by the dot product. Thus, finding the CCS is analogous to the problem faced in *partially observable Markov decision processes* (POMDPs) (Feng & Zilberstein, 2004), where optimal α-vectors (corresponding to the value vectors $\mathbf{u}(\mathbf{a})$) for all beliefs (corresponding to the weight vectors $\mathbf{w}$) must be found. Therefore, we can employ pruning operators from POMDP literature.

Algorithm 2 describes our implementation of `CPrune`, which is based on that of Feng and Zilberstein (2004) with one modification. In order to improve runtime guarantees, `CPrune` first pre-prunes the candidate solutions $\mathcal{U}$ to a PCS using the `PPrune` (Algorithm 3) at line 1. `PPrune` computes a PCS in $O(d|\mathcal{U}||PCS|)$ by running pairwise comparisons. Next, a partial CCS, $\mathcal{U}^*$, is constructed as follows: a random vector $\mathbf{u}$ from $\mathcal{U}$ is selected at line 4. For $\mathbf{u}$ the algorithm tries to find a weight vector $\mathbf{w}$ for which $\mathbf{u}$ is better than the vectors in $\mathcal{U}^*$ (line 5), by solving the linear program in Algorithm 4. If there is such a $\mathbf{w}$, `CPrune` finds the best vector $\mathbf{v}$ for $\mathbf{w}$ in $\mathcal{U}$ and moves it to $\mathcal{U}^*$ (line 11–13). If there is no weight for which $\mathbf{u}$ is better it is C-dominated and thus removed $\mathbf{u}$ from $\mathcal{U}$ (line 8).

Algorithm 2: `CPrune`($\mathcal{U}$)

```
   Input: A set of payoff vectors U
1  U ← PPrune(U)
2  U* ← ∅
3  while notEmpty(U) do
4      select random u from U
5      w ← findWeight(u,U*)
6      if w=null then
7          //did not find a weight where u is optimal
8          remove u from U
9      end
10     else
11         v ← arg max_{u∈U} w · u
12         U ← U \ {v}
13         U* ← U* ∪ {v}
14     end
15 end
16 return U*
```

The runtime of `CPrune` as defined by Algorithm 2 is

$$O(d|\mathcal{U}||PCS| + |PCS|P(d|CCS|)), \quad (3)$$

where $P(d|CCS|)$ is a polynomial in the size of the CCS and the number of objectives d, which is the runtime of the linear program that tests for C-domination (Algorithm 4).

Algorithm 3: `PPrune`($\mathcal{U}$)

Input: A set of payoff vectors $\mathcal{U}$

$\mathcal{U}^* \leftarrow \emptyset$
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* $\mathbf{u}$
    end
  end
  Remove $\mathbf{u}$, and all vectors P-dominated by $\mathbf{u}$, from $\mathcal{U}$
  Add $\mathbf{u}$ to $\mathcal{U}^*$
end
return $\mathcal{U}^*$

Algorithm 4: `findWeight`($\mathbf{u}$,$\mathcal{U}$)

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

if x > 0 return w else return null

The key downside of the non-graphical approach is that it requires explicitly enumerating all possible joint actions and calculating the payoffs associated with each one. Consequently, it is intractable for all but small numbers of agents, as the number of joint actions grows exponentially in the number of agents.

Theorem 3. *The time complexity of computing a CCS of a MO-CoG containing ρ local payoff functions, following the non-graphical approach (Equation 1) is:*

$$O(d\rho|\mathcal{A}_{max}|^n + d|\mathcal{A}_{max}|^n|PCS| + |PCS|P(d|CCS|)$$

Proof. First, $\mathcal{V}$ is computed by looping over all ρ VSFs for each joint action $\mathbf{a}$, summing vectors of length d. If the maximum size of the action space of an agent is $\mathcal{A}_{max}$ there are $O(|\mathcal{A}_{max}|^n)$ joint actions. $\mathcal{V}$ contains one payoff vector for each joint action. $\mathcal{V}$ is the input of `CPrune`. □

In the next two sections, we present two approaches to compute CCSs more efficiently. The first approach pushed the `CPrune` operator in Equation 1 into the cross-sum and union, just as the max-operator is pushed into the summation in VE. We call this the *inner loop approach*, as it uses pruning operators during agent eliminations, which is the inner loop of the VE algorithm. The second approach is inspired by *linear support* (Cheng, 1988), a POMDP pruning operator that only requires finding the optimal solution for certain $\mathbf{w}$. Instead of performing maximization over the entire set $\mathcal{V}$, as in the original linear support algorithm, we show that we can use VE on a finite number of *scalarized* instances of the MO-CoG, avoiding explicit calculation of $\mathcal{V}$. We call this approach the *outer loop approach*, as this it creates an outer loop around a single objective method (like VE), which it calls as a subroutine.

4. Convex Variable Elimination for MO-CoGs

In this section we show how to exploit loose couplings and calculate a CCS using an inner loop approach, i.e., by pushing the pruning operators into the cross-sum and union operators of Equation 1. The result is CMOVE, an extension to Rollón and Larrosa's Pareto-based extension of VE, which we refer to as PMOVE (Rollón & Larrosa, 2006). By analyzing CMOVE's complexity in terms of *local convex coverage sets*, we show that this approach yields much better runtime complexity guarantees than the non-graphical approach to computing CCSs that was presented in Section 3.

4.1 Exploiting Loose Couplings in the Inner Loop

In the non-graphical approach, computing a CCS is more expensive than computing a PCS, as we have shown in Section 3. We now show that, in MO-CoGs, we can compute a CCS much more efficiently by exploiting the MO-CoG's graphical structure. In particular, like in VE, we can solve the MO-CoG as a series of local subproblems, by *eliminating* agents and manipulating the set of VSFs $\mathcal{F}$ which describe the MO-CoG. The key idea is to compute *local CCSs (LCCSs)* when eliminating an agent instead of a single best response (as in VE). When computing an LCCS, the algorithm prunes away as many vectors as possible. This minimizes the number of payoff vectors that are calculated at the global level, which can greatly reduce computation time. Here we describe the `elim` operator for eliminating agents used by CMOVE in Section 4.2.

We first need to update our definition of neighboring local payoff functions (Definition 2), to neighboring VSFs.

Definition 10. *The set of neighboring VSFs $\mathcal{F}_i$ of i is the set of all local payoff functions that have agent i in scope.*

The neighboring agents n_i of an agent i are now the agents in the scope of a VSF in $\mathcal{F}_i$, except for i itself, corresponding to Definition 3. For each possible local joint action of n_i, we now compute an LCCS that contains the payoffs of the C-undominated responses of agent i, as the best response values of 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 $\mathbf{a}_{n_i}$. To compute the LCCS, we must consider all payoff vectors of the subproblem, $\mathcal{V}_i$, and prune the dominated ones.

Definition 11. *If we fix all actions in $\mathbf{a}_{n_i}$, but not 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 $\mathbf{a}_{n_i}$.*

Using Definition 11, we can now define the LCCS as the CCS of $\mathcal{V}_i$.

Definition 12. *A local CCS, an LCCS, is the C-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 create a new VSF, f^{new}, conditioned on the actions of the agents in n_i:

$$\forall \mathbf{a}_{n_i} \; f^{new}(\mathbf{a}_{n_i}) = LCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i}).$$

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

Proof. We show this by using the implication of Equation 2, i.e., for all joint actions $\mathbf{a}$ for which there is a $\mathbf{w}$ at which the scalarized value of $\mathbf{a}$ is maximal, a vector-valued payoff $\mathbf{u}(\mathbf{a}')$ for which $\mathbf{w} \cdot \mathbf{u}(\mathbf{a}') = \mathbf{w} \cdot \mathbf{u}(\mathbf{a}')$ is in the CCS. We show that the maximal scalarized payoff cannot be lost as a result of `elim`.

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 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 Equation 2, 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 scalarized value for all $\mathbf{w}$ and thereby preserving the CCS. □

Instead of an LCCS, we could compute a *local PCS (LPCS)*, that is, using a PCS computation on $\mathcal{V}_i$ instead of a CCS computation. Note that, since LCCS $\subseteq$ LPCS $\subseteq \mathcal{V}_i$, `elim` not only reduces the problem size with respect to $\mathcal{V}_i$, it can do so more than would be possible if we only considered P-dominance. Therefore, focusing on the CCS can greatly reduce the sizes of local subproblems. Since the solution of a local subproblem is the input for the next agent elimination, the size of subsequent local subproblems is also reduced, which can lead to considerable speed-ups.

4.2 Convex Multi-objective Variable Elimination

We now present the *convex multi-objective variable elimination (CMOVE)* algorithm, which implements `elim` using `CPrune`. Like VE, CMOVE iteratively eliminates agents until none are left. However, our implementation of `elim` computes a CCS and outputs the correct joint actions for each payoff vector in this CCS, rather than a single joint action. CMOVE is an extension to Rollón and Larrosa's Pareto-based extension of VE, which we refer to as PMOVE (Rollón & Larrosa, 2006).

The most important difference between CMOVE and PMOVE is that CMOVE computes a CCS, which typically leads to much 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 the *tagging scheme* instead of the *backwards pass*, as in Section 2.2.

Algorithm 5 presents an abstract version of CMOVE that leaves the pruning operators unspecified. As in Section 3, CMOVE first translates the problem into a set of *vector-set factors* (VSFs), $\mathcal{F}$ on line 1. Next, CMOVE iteratively eliminates agents using `elim` (line 2–5). The elimination order can be determined using techniques devised for single-objective VE (Koller & Friedman, 2009).

Algorithm 5: `CMOVE`($\mathcal{U}$, `prune1`, `prune2`, `prune3`, `q`)

Input: A set of local payoff functions $\mathcal{U}$ and an elimination order `q` (a queue containing all agents)

1 $\mathcal{F} \leftarrow$ create one VSF for every local payoff function in $\mathcal{U}$
2 **while** $\mathbf{a}_{n_i} \in \mathcal{A}_{n_i}$ **do**
3 | $i \leftarrow$ `q.dequeue()`
4 | $\mathcal{F} \leftarrow$ `elim`($\mathcal{F}, i,$ `prune1`, `prune2`)
5 **end**
6 $f \leftarrow$ retrieve final factor from $\mathcal{F}$
7 $\mathcal{S} \leftarrow f(a_\emptyset)$
8 **return** `prune3`($\mathcal{S}$)

Algorithm 6 shows our implementation of `elim`, parameterized with two pruning operators, `prune1` and `prune2`, corresponding to two different pruning locations inside the operator that computes $LCCS_i$: $ComputeLCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i},$ `prune1`, `prune2`$)$.

Algorithm 6: `elim`($\mathcal{F}, i,$ `prune1`, `prune2`)

Input: A set of VSFs $\mathcal{F}$, and an agent i

$n_i \leftarrow$ the set of neighboring agents of i
$\mathcal{F}_i \leftarrow$ the subset of VSF that have i in scope
$f^{new}(\mathbf{a}_{n_i}) \leftarrow$ a new VSF
foreach $\mathbf{a}_{n_i} \in \mathcal{A}_{n_i}$ **do**
| $f^{new}(\mathbf{a}_{n_i}) \leftarrow ComputeLCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i},$ `prune1`, `prune2`$)$
end
$\mathcal{F} \leftarrow \mathcal{F} \setminus \mathcal{F}_i \cup \{f^{new}\}$
return $\mathcal{F}$

$ComputeLCCS_i$ is implemented as follows: first we define a new cross-sum-and-prune operator $A\hat{\oplus}B = \texttt{prune1}(A \oplus B)$. $LCCS_i$ applies this operator sequentially:

$$ComputeLCCS_i(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2}) = \texttt{prune2}(\bigcup_{a_i} \hat{\bigoplus}_{f^e \in \mathcal{F}_i} f^e(\mathbf{a}_e)). \tag{4}$$

`prune1` is applied to each cross-sum of two sets, via the $\hat{\oplus}$ operator, leading to *incremental pruning* (Cassandra, Littman, & Zhang, 1997). `prune2` is applied at a coarser level, after the union. CMOVE applies `elim` iteratively until no agents remain, resulting in a CCS. Note that, when there are no agents left, f^{new} on line 3 has no agents to condition on. In this case, we consider the "actions of the neighbors" to be a single empty action: $a_\emptyset$.

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`), as reflected in Algorithm 5. After all agents have been eliminated, the final factor is taken from the set of factors (line 6), and the single set, S contained in that factor is retrieved (line 7). Note that we use the empty action $a_\emptyset$ to denote the field in the final factor, as it has no agents in scope. Finally `prune3` is called on S.

Consider the example in Figure 2a, using the payoffs defined by Table 2, and apply CMOVE. First, CMOVE creates the VSFs f^1 and f^2 from u^1 and u^2. To eliminate agent 3, it creates a new VSF $f^3(a_2)$ 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 $f^3(\bar{a}_2) = \{(1,1)_{\bar{a}_3}\}$. CMOVE then adds f^3 to the graph and removes f^2 and agent 3, yielding the factor graph shown in Figure 2b.

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 yields $\{(7,2)_{\dot{a}_2\dot{a}_3}, (5,4)_{\dot{a}_2\bar{a}_3}\}$ and the second yields $\{(1,1)_{\bar{a}_2\bar{a}_3}\}$. 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 graph in Figure 2c.

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 $CCS = \{(7,2)_{\{\dot{a}_1\dot{a}_2\dot{a}_3\}}, (4,7)_{\{\bar{a}_1\bar{a}_2\bar{a}_3\}}\}$.

4.3 CMOVE Variants

There are several ways to implement the pruning operators that lead to correct instantiations of CMOVE. Both `PPrune` (Algorithm 2) and `CPrune` (Algorithm 1) can be used, as long as either `prune2` or `prune3` is `CPrune`. Note that if `prune2` computes the CCS, `prune3` is not necessary.

In this article, 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`. The latter invests more effort in intermediate pruning, which can result in smaller cross-sums, and a resulting speedup. However, when only a few vectors can be pruned in these intermediate steps, this additional speedup may not occur, and the algorithm creates unnecessary overhead.[7] We empirically investigate these variants in Section 4.5

One could also consider using pruning operators that contain prior knowledge about the range of possible weight vectors. If such information is available, it could be easily incorporated by changing the pruning operators accordingly, leading to even smaller LCCSs, and thus a faster algorithm. In this article however, we focus on the case in which such prior knowledge is not available.

4.4 Analysis

We now analyze the correctness and complexity of CMOVE.

Theorem 5. *MOVE correctly computes a 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 6. *The computational complexity of CMOVE is*

$$O(\ n\ |\mathcal{A}_{max}|^{w_a}\ (w_f\ R_1 + R_2) + R_3\ ), \tag{5}$$

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 an LCCS for each joint action of the eliminated agent's neighbors, in a field in a new VSF. CMOVE computes $O(|\mathcal{A}_{max}|^{w_a})$ fields per iteration, calling `prune1` (Equation 4) for each adjacent factor, and `prune2` once after taking the union over actions of the eliminated agent. `prune3` is called exactly once, after eliminating all agents (line 8 of Algorithm 5). □

Unlike the non-graphical approach, CMOVE is exponential only in w_a, not the number of agents. In this respect, our results are similar to those for PMOVE (Rollón, 2008). However, those earlier complexity results do not make the effect of pruning explicit. Instead, the complexity bound makes use of additional problem constraints, which limit the total number of possible different value vectors. Specifically, in the analysis of PMOVE, the payoff vectors are integer-valued, with a maximum value for all objectives. In practice,

7. We can also compute a PCS first, using `prune1` and `prune2`, and then compute the CCS with `prune3`. However, this is useful only for small problems for which a PCS is cheaper to compute than a CCS.

such bounds can be very loose or even impossible to define (e.g., when the payoff values are real-valued in one or more objectives). 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 4.5.

Theorem 6 demonstrates that the complexity of CMOVE depends heavily 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`. We therefore need to balance the effort of the lower-level pruning with that of the higher-level pruning, which occurs less often but is dependent on the output of the lower level. The bigger the LCCSs, the more can be gained from lower-level pruning.

Theorem 7. *The space complexity of CMOVE is*

$$O(\; d\; n\; |\mathcal{A}_{max}|^{w_a}\; |LCCS_{max}| + \; d\; \rho\; |\mathcal{A}_{max}|^{|e_{max}|}\;),$$

where $|LCCS_{max}|$ *is maximum size of a local CCS,* ρ *is the original number of VSFs, and* $|e_{max}|$ *is the maximum scope size of the original VSFs.*

Proof. CMOVE computes a local CCS for each new VSF for each joint action of the eliminated agent's neighbors. There are maximally w_a neighbors. There are maximally n new factors. Each payoff vector stores d real numbers.

There are ρ VSFs created during the initialization of CMOVE. All of these VSFs have exactly one payoff vector containing d real numbers, per joint action of the agents in scope. There are maximally $|\mathcal{A}_{max}|^{|e_{max}|}$ such joint actions. □

For PMOVE, the space complexity is the same but with $|PCCS_{max}|$ instead of $|LCCS_{max}|$. Because the LCCS is a subset of the corresponding LPCS, CMOVE is thus strictly more memory efficient than PMOVE.

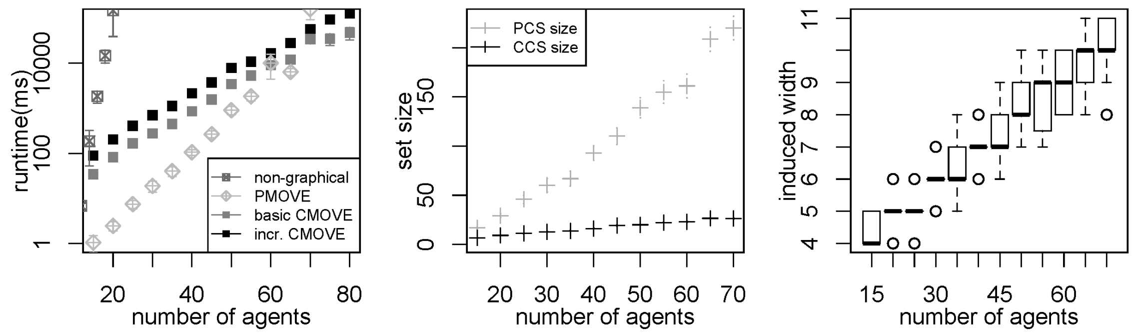
Note that Theorem 7 is a rather loose upper bound on the space complexity, as not all VSFs, original or new, exist at the same time. However, it is not possible to to predict a priori how many of these VSFs exist at the same time, resulting in a space complexity bound on the basis of all VSFs that exist at some point during the execution of CMOVE.

4.5 Empirical Evaluation

To test the efficiency of CMOVE, we now compare its runtimes to those of PMOVE[8] and the non-graphical approach for problems with varying numbers of agents and objectives. We also analyze how these runtimes correspond to the sizes of the PCS and CCS.

We use two types of experiments. The first experiments are done with random MO-CoGs in which we can directly control all variables. In the second experiment, we use Mining Day, a more realistic benchmark, that is more structured than random MO-CoGs but still randomized.

8. We compare to PMOVE using only `prune2` = `PPrune`, rather than `prune1` = `prune2` = `PPrune`, as was proposed in the original article (Rollón & Larrosa, 2006) because we found the former option slightly but consistently faster.



(a) (b) (c)

Figure 4: (a) Runtimes (ms) in log-scale for the nongraphical method, PMOVE and CMOVE with standard deviation of mean (error bars), (b) the corresponding number of vectors in the PCS and CCS, and (c) the corresponding spread of the induced width.

4.5.1 Random Graphs

To generate random MO-CoGs, we employ a procedure that takes as input: 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. The procedure then starts with a fully connected graph with local payoff functions connecting to two agents each. Then, local payoff functions are randomly removed, while ensuring that the graph remains connected, until only ρ local payoff functions remain. The values for the different objectives in each local payoff function are real numbers that are drawn independently and uniformly from the interval $[0, 10]$. We compare algorithms on the same set of randomly generated MO-CoGs for each separate value of n, d, ρ, and $|\mathcal{A}_i|$.

To compare basic CMOVE, incremental CMOVE, PMOVE, and the non-graphical method, we test them on random MO-CoGs with the number of agents ranging between 10 and 85, the average number of factors per agent held at $\rho = 1.5n$, and the number of objectives $d = 2$. This experiment was run on a 2.4 GHz Intel Core i5 computer, with 4 GB memory. Figure 4 shows the results, averaged over 20 MO-CoGs for each number of agents. The runtime (Figure 4a) of the non-graphical method quickly explodes. Both CMOVE variants are slower than PMOVE for small numbers of agents, but the runtime grows much more slowly than that of PMOVE. At 70 agents, both CMOVE variants are faster than PMOVE on average. For 75 agents, one of the MO-CoGs generated caused PMOVE to time out at $5000s$, while basic CMOVE had a maximum runtime of $132s$, and incremental CMOVE $136s$. This can be explained by the differences in the size of the solutions, i.e., the PCS and the CCS (Figure 4b). The PCS grows much more quickly with the number of agents than the CCS does. For two-objective problems, incremental CMOVE seems to be consistently slower than basic CMOVE.

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. This can be explained by the induced width of the MO-CoGs, in which the runtime of CMOVE is exponential. In Figure 4c, we see that the induced width increases linearly with the number of agents for random graphs.

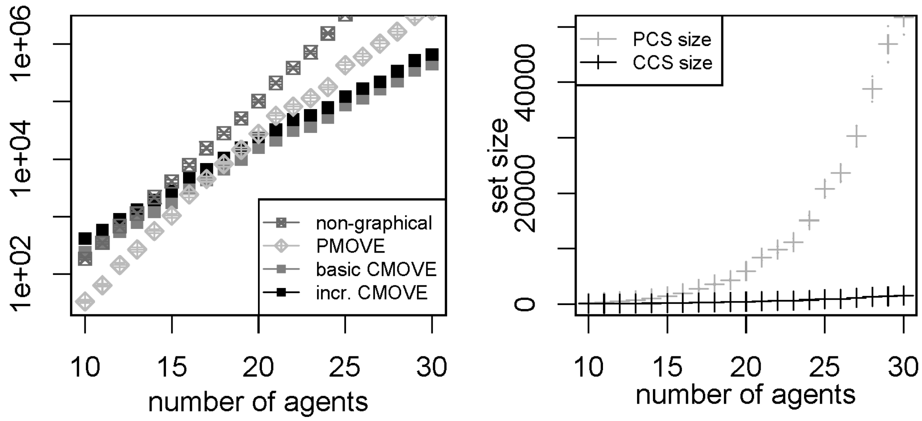


Figure 5: Runtimes (ms) for the non-graphical method, PMOVE and CMOVE in log-scale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of agents and 5 objectives.

We therefore conclude that, in two-objective MO-CoGs, the non-graphical method is intractable, even for small numbers of agents, and that the runtime of CMOVE increases much less with the number of agents than PMOVE does.

To test how the runtime behavior changes with a higher number of objectives, we run the same experiment with the average number of factors per agent held at $\rho = 1.5n$ and increasing numbers of agents again, but now for $d = 5$. This and all remaining experiments described in this section were executed on a Xeon L5520 2.26 GHz computer with 24 GB memory. Figure 5 (left) shows the results of this experiment, averaged over 85 MO-CoGs for each number of agents. Note that we do not plot the induced widths, as this does not change with the number of objectives. These results demonstrate that, as the number of agents grows, using CMOVE 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.

As before, the runtime of 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 random MO-CoG generation procedure. However, CMOVE's runtime is polynomial in the size of the CCS, and this size grows exponentially, as shown in Figure 5 (right). 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 6 (left) 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

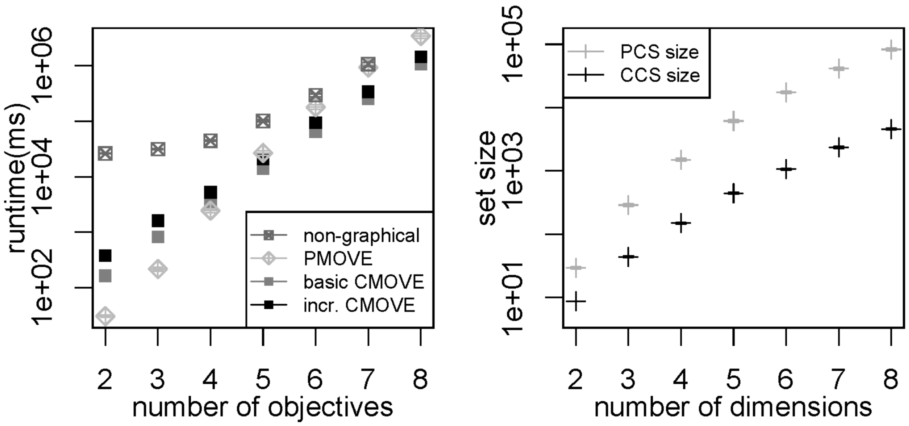


Figure 6: Runtimes (ms) for the non-graphical method, PMOVE and CMOVE in logscale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of objectives.

several orders of magnitude slower at $d = 2$, grows slowly until $d = 5$, and then starts to grow with about the same exponent as PMOVE. This can be explained by the fact that the time it takes to enumerate of all joint actions and payoffs remains approximately constant, while the time it takes to prune increases exponentially with the number of objectives. 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 8 dimensions they are respectively 3.2 and 2.4 times faster. This happens because the CCS grows much more slowly than the PCS, as shown in Figure 6 (right). 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 $d = 8$. This trend indicates that pruning after every cross-sum, i.e., at `prune1`, becomes (relatively) better for higher numbers of objectives. Although we were unable to solve problem instances with many more objectives within reasonable time, we expect this trend to continue and that incremental CMOVE would be faster than basic CMOVE for problems with very many objectives.

Overall, we conclude that, for random graphs, CMOVE is key to solving MO-CoGs within reasonable time, especially when the problem size increases in either the number of agents, the number of objectives, or both.

4.5.2 Mining Day

In Mining Day, a mining company mines gold and silver (objectives) from a set of mines (local payoff functions) located in the mountains (see Figure 1). The mine workers live in villages at the foot of the mountains. 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 fluctuating prices of gold and silver. To maximize revenue,

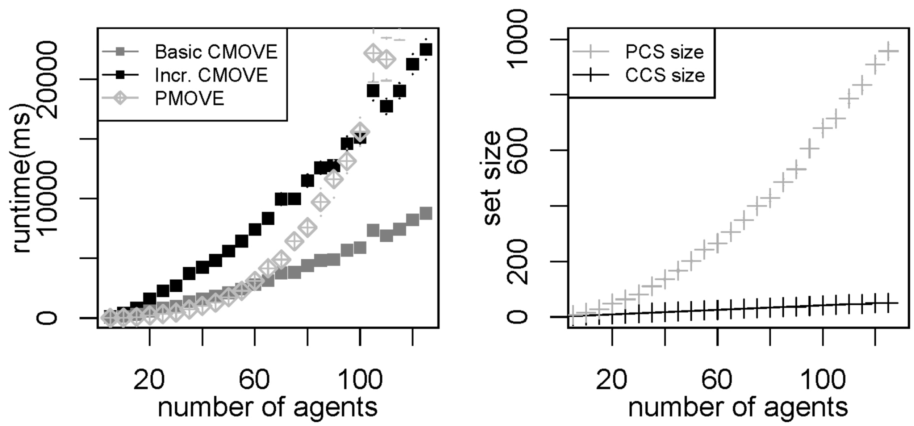


Figure 7: Runtimes (ms) for basic and incremental CMOVE, and PMOVE, in log-scale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of agents.

the mining company wants to use the latest possible price information, and not lose time recomputing the optimal strategy with every price change. Therefore, we must calculate a CCS.

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 the interval $[0, 10]$.

In order to compare the runtimes of basic and incremental CMOVE against PMOVE on a more realistic benchmark, we generate Mining Day instances with varying numbers of agents. Note that we do not include the non-graphical method, as its runtime mainly depends on the number of agents, and is thus not considerably faster for this problem than for random graphs. The runtime results are shown in Figure 7 (left). Both CMOVE and PMOVE are able to tackle problems with over 100 agents. However, the runtime of PMOVE grows much more quickly than that of CMOVE. In this two-objective 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 much more slowly than that of PMOVE, and it 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, as shown in Figure 7 (right). The induced width remains around 4 regardless of the number of agents. These results demonstrate that, as the CCS grows more slowly than the PCS with the number of agents, CMOVE can solve MO-CoGs more efficiently than PMOVE as the number of agents increases.

5. Linear Support for MO-CoGs

In this section, we present *variable elimination linear support* (VELS). VELS is a new method for computing the CCS in MO-CoGs that has several advantages over CMOVE: for moderate numbers of objectives, its runtime complexity is better; it is an *anytime* algorithm, i.e., over time, VELS produces intermediate results which become better and better approximations of the CCS and therefore, when provided with a maximum scalarized error ε, VELS can compute an ε-optimal CCS.

Rather than dealing with the multiple objectives in the inner loop (like CMOVE), VELS deals with them in the *outer loop* and employs VE as a subroutine. VELS thus builds the CCS incrementally. With each iteration of its outer loop, VELS adds at most one new vector to a partial CCS. To find this vector, VELS selects a single $\mathbf{w}$ (the one that offers the maximal possible improvement), and passes that $\mathbf{w}$ to the inner loop. In the inner loop, VELS uses VE (Section 2.2) to solve the single-objective *coordination graph* (CoG) that results from scalarizing the MO-CoG using the $\mathbf{w}$ selected by the outer loop. The joint action that is optimal for this CoG and its multi-objective payoff are then added to the partial CCS.

The departure point for creating VELS is *Cheng's linear support* (Cheng, 1988). Cheng's linear support was originally designed as a pruning algorithm for POMDPs. Unfortunately, this algorithm is rarely used for POMDPs in practice, as its runtime is exponential in the number of states. However, the number of states in a POMDP corresponds to the number of objectives in a MO-CoG, and while realistic POMDPs typically have many states, many MO-CoGs have only a handful of objectives. Therefore, for MO-CoGs, the scalability in the number of agents is more important, making Cheng's linear support an attractive starting point for developing an efficient MO-CoG solution method.

Building on Cheng's linear support, in Section 5.1 we create an abstract algorithm that we call *optimistic linear support* (OLS), which builds up the CCS incrementally. Because OLS takes an arbitrary single-objective problem solver as input, it can be seen as a generic multi-objective method. We show that OLS chooses a $\mathbf{w}$ at each iteration such that, after a finite number of iterations, no further improvements to the partial CCS can be made and OLS can terminate. Furthermore, we bound the maximum scalarized error of the intermediate results, so that they can be used as bounded approximations of the CCS. Then, in Section 5.2, we instantiate OLS by using VE as its single-objective problem solver, yielding VELS, an effective MO-CoG algorithm.

5.1 Optimistic Linear Support

OLS constructs the CCS incrementally, by adding vectors to an initially empty *partial CCS*:

Definition 13. *A* partial CCS*, S, is a subset of the CCS, which is in turn a subset of $\mathcal{V}$: $S \subseteq CCS \subseteq \mathcal{V}$.*

We define the scalarized value function over S, corresponding to the convex upper surface (shown in bold) in Figure 8b-d:

Definition 14. *A scalarized value function over a partial CCS, S, is a function that takes a weight vector $\mathbf{w}$ as input, and returns the maximal attainable scalarized value with any*

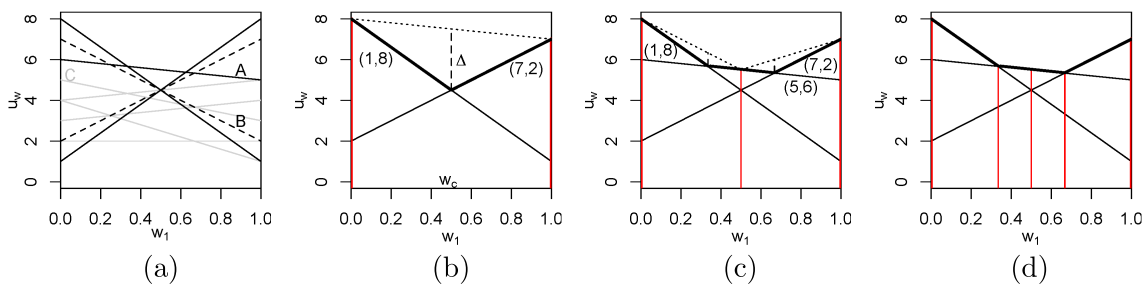


Figure 8: (a) All possible payoff vectors for a 2-objective MO-CoG. (b) OLS finds two payoff vectors at the extrema (red vertical lines), a new corner weight $\mathbf{w}_c = (0.5, 0.5)$ is found, with maximal possible improvement Δ. $\overline{CCS}$ is shown as the dotted line. (c) OLS finds a new vector at $(0.5, 0.5)$, and adds two new corner weights to Q. (d) OLS calls `SolveCoG` for both corner weights (in two iterations), and finds no new vectors, ensuring $S = \overline{CCS} = CCS$.

payoff vector in S:

$$u^*_S(\mathbf{w}) = \max_{\mathbf{u}(\mathbf{a}) \in S} \mathbf{w} \cdot \mathbf{u}(\mathbf{a}).$$

Similarly, we define the set of maximizing joint actions:

Definition 15. *The optimal joint action set function with respect to S is a function that gives the joint actions that maximize the scalarized value:*

$$A_S(\mathbf{w}) = \arg\max_{\mathbf{u}(\mathbf{a}) \in S} \mathbf{w} \cdot \mathbf{u}(\mathbf{a}).$$

Note that $A_S(\mathbf{w})$ is a set because for some $\mathbf{w}$ there can be multiple joint actions that provide the same scalarized value.

Using these definitions, we can describe *optimistic linear support (OLS)*. OLS adds vectors to a partial CCS, S, finding new vectors for so-called *corner weights*. These corner weights are the weights where $u^*_S(\mathbf{w})$ (Definition 14) changes slope in all directions. These must thus be weights where $A_S(\mathbf{w})$ (Definition 15) consists of multiple payoff vectors. Every corner weight is prioritized by the maximal possible improvement of finding a new payoff vector at that corner weight. When the maximal possible improvement is 0, OLS knows that the partial CCS is complete. An example of this process is given in Figure 8, where the (corner) weights where the algorithm has searched for new payoff vectors are indicated by red vertical lines.

OLS is shown in Algorithm 7. To find the optimal payoff for a corner weight, OLS assumes access to a function called `SolveCoG` that computes the best payoff vector for a given $\mathbf{w}$. For now, we leave the implementation of `SolveCoG` abstract. In Section 5.2, we discuss how to implement `SolveCoG`. OLS also takes as input m, the MO-CoG to be solved, and ε, the maximal tolerable error in the result.

We first describe how OLS is initialized (Section 5.1.1). Then, we define corner weights formally and describe how OLS identifies them (Section 5.1.2). Finally, we describe how

Algorithm 7: `OLS`$(m,$ `SolveCoG`$, \varepsilon)$

1 $S \leftarrow \emptyset$//partial CCS,
2 $\mathcal{W} \leftarrow \emptyset$ //set of checked weights
3 $Q \leftarrow$ an empty priority queue
4 **foreach** *extremum of the weight simplex* $\mathbf{w}_e$ **do**
5   Q.add($\mathbf{w}_e$, ∞) // add extrema with infinite priority
6 **end**
7 **while** $\neg$*Q.isEmpty()* $\wedge$ $\neg$*timeOut* **do**
8     $\mathbf{w} \leftarrow$ Q.pop()
9     $\mathbf{u} \leftarrow$ `SolveCoG`$(m, \mathbf{w})$
10   **if** $\mathbf{u} \notin S$ **then**
11         $W_{del} \leftarrow$ remove the corner weights made obsolete by $\mathbf{u}$ from Q, and store them
12         $W_{del} \leftarrow \{\mathbf{w}\} \cup W_{del}$ //corner weights which are removed because of adding $\mathbf{u}$
13         $W_{\mathbf{u}} \leftarrow$ `newCornerWeights`$(\mathbf{u}, W_{del}, S)$
14         $S \leftarrow S \cup \{\mathbf{u}\}$
15   **foreach** $\mathbf{w} \in W_{\mathbf{u}}$ **do**
16             $\Delta_r(\mathbf{w}) \leftarrow$ calculate improvement using `maxValueLP`$(\mathbf{w}, S, \mathcal{W})$
17   **if** $\Delta_r(\mathbf{w}) > \varepsilon$ **then**
18   Q.add($\mathbf{w}$, $\Delta_r(\mathbf{w})$)
19   **end**
20   **end**
21   **end**
22     $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathbf{w}\}$
23 **end**
24 **return** S *and the highest* $\Delta_r(\mathbf{w})$ *left in* Q

OLS prioritizes corner weights and how this can also be used to bound the error when stopping OLS before it is done finding a full CCS (Section 5.1.3).

5.1.1 Initialization

OLS starts by initializing the partial CCS, S, which will contain the payoff vectors in the CCS discovered so far (line 1 of Algorithm 7), as well as the set of visited weights $\mathcal{W}$ (line 2). Then, it adds the extrema of the weight simplex, i.e., those points where all of the weight is on one objective, to a priority queue Q, with infinite priority (line 5).

These extrema are popped off the priority queue when OLS enters the main loop (line 7), in which the $\mathbf{w}$ with the highest priority is selected (line 8). `SolveCoG` is then called with $\mathbf{w}$ (line 9) to find $\mathbf{u}$, the best payoff vector for that $\mathbf{w}$.

For example, Figure 8b shows S after two payoff vectors of a 2-dimensional MO-CoG have been found by applying `SolveCoG` to the extrema of the weight simplex: $S = \{(1, 8), (7, 2)\}$. Each of these vectors must be part of the CCS because it is optimal for at least one $\mathbf{w}$: the one for which `SolveCoG` returned it as a solution (the extrema of the weight simplex). The set of weights $\mathcal{W}$ that OLS has tested so far are marked with vertical red line segments.

5.1.2 Corner Weights

After having evaluated the extrema, S consists of d (the number of objectives) payoff vectors and associated joint actions. However, for many weights on the simplex, it does not yet contain the optimal payoff vector. Therefore, after identifying a new vector $\mathbf{u}$ to add to S (line 9), OLS must determine what new weights to add to Q. Like Cheng's linear support, OLS does so by identifying the *corner weights*: the weights at the corners of the convex upper surface, i.e., the points where the PWLC surface $u^*_S(\mathbf{w})$ changes slope. To define the corner weights precisely, we must first define P, the polyhedral subspace of the weight simplex that is above $u^*_S(\mathbf{w})$ (Bertsimas & Tsitsiklis, 1997). The corner weights are the vertices of P, which can be defined by a set of linear inequalities:

Definition 16. *If S is the set of known payoff vectors, we define a polyhedron*

$$P = \{\mathbf{x} \in \Re^{d+1} : \mathcal{S}^{+}\mathbf{x} \geq \vec{0}, \forall i, w_i > 0, \sum_i w_i = 1\},$$

where $\mathcal{S}^+$ is a matrix with the elements of S as row vectors, augmented by a column vector of -1's. The set of linear inequalities $\mathcal{S}^+\mathbf{x} \geq \vec{0}$, is supplemented by the simplex constraints: $\forall i\ w_i > 0$ and $\sum_i w_i = 1$. The vector $\mathbf{x} = (w_1, ..., w_d, u)$ consists of a weight vector and a scalarized value at those weights. The corner weights *are the weights contained in the vertices of P, which are also of the form $(w_1, ..., w_d, u)$.*

Note that, due to the simplex constraints, P is only d-dimensional. Furthermore, the extrema of the weight simplex are special cases of corner weights.

After identifying $\mathbf{u}$, OLS identifies which corner weights change in the polyhedron P by adding $\mathbf{u}$ to S. Fortunately, this does not require recomputation of all the corner weights, but can be done incrementally: first, the corner weights in Q for which $\mathbf{u}$ yields a better value than currently known are deleted from the queue (line 11) and then the function $\texttt{newCornerWeights}(\mathbf{u}, W_{del}, S)$ at line 13 calculates the new corner weights that involve $\mathbf{u}$ by solving a system of linear equations to see where $\mathbf{u}$ intersects with the boundaries and the relevant subset of the present vectors in S.

$\texttt{newCornerWeights}(\mathbf{u}, W_{del}, S)$ (line 13) first calculates the set of all relevant payoff vectors, A_{rel}, by taking the union of all the maximizing vectors of the weights in W_{del}[9]:

$$A_{rel} = \bigcup_{\mathbf{w} \in W_{del}} A_S(\mathbf{w}).$$

If any $A_S(\mathbf{w})$ contains fewer than d payoff vectors, then a boundary of the weight simplex is involved. These boundaries are also stored. All possible subsets of size $d-1$ (of vectors and boundaries) are taken. For each subset the weight where these $d-1$ payoff vectors (and/or boundaries) intersect with each other and $\mathbf{u}$ is computed by solving a system of linear equations. The intersection weights for all subsets together form the set of candidate corner weights: W_{can}. $\texttt{newCornerWeights}(\mathbf{u}, W_{del}, S)$ returns the subset of W_{can} which are inside of the weight simplex and for which $\mathbf{u}$ has a higher scalarized value than any payoff

9. In fact, in our implementation, we optimize this step by caching $A_S(\mathbf{w})$ for all $\mathbf{w}$ in Q.

vector already in S. Figure 8b shows one new corner weight labelled $\mathbf{w}_c = (0.5, 0.5)$. In practice, $|A_{rel}|$ is very small, so only a few systems of linear equations need to be solved.[10]

After calculating the new corner weights $W_\mathbf{u}$ at line 13, $\mathbf{u}$ is added to S at line 14. Cheng showed that finding the best payoff vector for each corner weight and adding it to the partial CCS, i.e., $S \leftarrow S \cup \{\texttt{SolveCoG}(\mathbf{w})\}$, guarantees the best improvement to S:

Theorem 8. *(Cheng 1988) The maximum value of:*

$$\max_{\mathbf{w},\mathbf{u} \in CCS} \min_{\mathbf{v} \in S} \mathbf{w} \cdot \mathbf{u} - \mathbf{w} \cdot \mathbf{v},$$

i.e., the maximal improvement to S by adding a vector to it, is at one of the corner weights (Cheng, 1988).

Theorem 8 guarantees the correctness of OLS: after all corner weights are checked, there are no new payoff vectors; thus the maximal improvement must be 0 and OLS has found the full CCS.

5.1.3 Prioritization

Cheng's linear support assumes that all corner weights can be checked inexpensively, which is a reasonable assumption in a POMDP setting. However, since `SolveCoG` is an expensive operation, testing all corner weights may not be feasible in MO-CoGs. Therefore, unlike Cheng's linear support, OLS pops only one $\mathbf{w}$ off Q to be tested per iteration. Making OLS efficient thus critically depends on giving each $\mathbf{w}$ a suitable priority when adding it to Q. To this end, OLS prioritizes each corner weight $\mathbf{w}$ according to its *maximal possible improvement*, an upper bound on the improvement in $u^*_S(\mathbf{w})$. This upper bound is computed with respect to $\overline{CCS}$, the *optimistic hypothetical CCS*, i.e., the best-case scenario for the final CCS given that S is the current partial CCS and $\mathcal{W}$ is the set of weights already tested with `SolveCoG`. The key advantage of OLS over Cheng's linear support is that these priorities can be computed without calling `SolveCoG`, obviating the need to run `SolveCoG` on all corner weights.

Definition 17. *An* optimistic hypothetical CCS, *$\overline{CCS}$ is a set of payoff vectors that yields the highest possible scalarized value for all possible $\mathbf{w}$ consistent with finding the vectors S at the weights in $\mathcal{W}$.*

Figure 8b denotes the $\overline{CCS} = \{(1, 8), (7, 2), (7, 8)\}$ with a dotted line. Note that $\overline{CCS}$ is a superset of S and the value of $u^*_{\overline{CCS}}(\mathbf{w})$ is the same as $u^*_S(\mathbf{w})$ at all the weights in $\mathcal{W}$. For a given $\mathbf{w}$, `maxValueLP` finds the the scalarized value of $u^*_{\overline{CCS}}(\mathbf{w})$ by solving:

$$\begin{aligned} &\max\ \mathbf{w} \cdot \mathbf{v} \\ &\text{subject to} \quad \mathcal{W}\ \mathbf{v} \leq \mathbf{u}^*_{S,\mathcal{W}}, \end{aligned}$$

10. However, in theory it is possible to construct a partial CCS, S that has a corner weight for which all payoff vectors in S are in A_{del}.

where $\mathbf{u}^*_{S,\mathcal{W}}$ is a vector containing $u^*_S(\mathbf{w}')$ for all $\mathbf{w}' \in \mathcal{W}$. Note that we abuse the notation $\mathcal{W}$, which in this case is a matrix whose rows consist of all the weight vectors in the set $\mathcal{W}$.[11]

Using $\overline{CCS}$, we can define the maximal possible improvement:

$$\Delta(\mathbf{w}) = u^*_{\overline{CCS}}(\mathbf{w}) - u^*_S(\mathbf{w}).$$

Figure 8b shows $\Delta(\mathbf{w}_c)$ with a dashed line. We use the *maximal relative possible improvement*, $\Delta_r(\mathbf{w}) = \Delta(\mathbf{w})/u^*_{\overline{CCS}}(\mathbf{w})$, as the priority of each new corner weight $\mathbf{w} \in W_{\mathbf{u}}$. In Figure 8b, $\Delta_r(\mathbf{w}_c)=\frac{(0.5,0.5)\cdot((7,8)-(1,8))}{7.5}=0.4$. When a corner weight $\mathbf{w}$ is identified (line 13), it is added to Q with priority $\Delta_r(\mathbf{w})$ as long as $\Delta_r(\mathbf{w}) > \varepsilon$ (lines 16-18).

After $\mathbf{w}_c$ in Figure 8b is added to Q, it is popped off again (as it is the only element of Q). `SolveCoG`($\mathbf{w}_c$) generates a new vector $(5, 6)$, yielding $S = \{(1, 8), (7, 2), (5, 6)\}$, as illustrated in Figure 8c. The new corner weights $(0.667, 0.333)$ and $(0.333, 0.667)$ are the points at which $(5, 6)$ intersects with $(7, 2)$ and $(1, 8)$. Testing these weights, as illustrated in Figure 8d, does not result in new payoff vectors, causing OLS to terminate. The maximal improvement at these corner weights is 0 and thus, due to Theorem 8, $S = CCS$ upon termination. OLS called `solveCoG` for only 5 weights resulting exactly in the 3 payoff vectors of the CCS. The other 7 payoff vectors in $\mathcal{V}$ (displayed as grey and dashed black lines in Figure 8a) were never generated.

5.2 Variable Elimination Linear Support

Any exact CoG algorithm can be used to implement `SolveCoG`. A naive approach is to explicitly compute the values of all joint actions $\mathcal{V}$ and select the joint action that maximizes this value:

$$\texttt{SolveCoG}(m, \mathbf{w}) = \arg\max_{\mathbf{u}(\mathbf{a})\in\mathcal{V}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a}).$$

This implementation of `SolveCoG` in combination with OLS yields an algorithm that we refer to as *non-graphical linear support (NGLS)*, because it ignores the graphical structure, flattening the CoG into a standard multi-objective cooperative normal form game. The main downside is that the computational complexity of `SolveCoG` is linear in $|\mathcal{V}|$ (which is equal to $|\mathcal{A}|$), which is exponential in the number of agents, making it feasible only for MO-CoGs with very few agents.

By contrast, if we use VE (Section 2.2) to implement `SolveCoG`, we can do better. We call the resulting algorithm *variable elimination linear support (VELS)*. Having dealt with the multiple objectives in the outer loop of OLS, VELS relies on VE to exploit the graphical structure in the inner loop, yielding a much more efficient method than NGLS.

5.3 Analysis

We now analyze the computational complexity of VELS.

11. Our implementation of OLS reduces the size of the LP by using only the subset of weights in $\mathcal{W}$ for which the joint actions involved in $\mathbf{w}$, $A_S(\mathbf{w})$, have been found to be optimal. This can lead to a slight overestimation of $u^*_{\overline{CCS}}(\mathbf{w})$.

Theorem 9. *The runtime of VELS with $\varepsilon = 0$ is*

$$O((|CCS| + |\mathcal{W}_{CCS}|)(n|\mathcal{A}_{max}|^w + C_{nw} + C_{heur})),$$

where w is the induced width when running VE, $|CCS|$ is the size of the CCS, $|\mathcal{W}_{CCS}|$ is the number of corner weights of $u^*_{CCS}(\mathbf{w})$, C_{nw} the time it costs to run `newCornerWeights`*, and C_{heur} the cost of the computation of the value of the optimistic CCS using* `maxValueLP`*.*

Proof. Since $n|\mathcal{A}_{max}|^w$ is the runtime of VE (Theorem 1), the runtime of VELS is this quantity (plus the overhead per corner weight $C_{nw} + C_{heur}$) multiplied by the number of calls to VE. To count these calls, we consider two cases: calls to VE that result in adding a new vector to S and those that do not result in a new vector but instead confirm the optimality of the scalarized value at that weight. The former is the size of the final CCS, $|CCS|$, while the latter is the number of corner weights for the final CCS, $|\mathcal{W}_{CCS}|$. □

The overhead of OLS itself, i.e., computing new corner weights, C_{nw}, and calculating the maximal relative improvement, C_{heur}, is very small compared to the `SolveCoG` calls. In practice, `newCornerWeights`($\mathbf{u}$, W_{del}, S) computes the solutions to only a small set of linear equations (of d equations each). `maxValueLP`($\mathbf{w}$, S, $\mathcal{W}$) computes the solutions to linear programs, which is polynomial in the size of its inputs.[12]

For $d = 2$, the number of corner weights is smaller than $|CCS|$ and the runtime of VELS is thus $O(n|\mathcal{A}_{max}|^w|CCS|)$. For $d = 3$, the number of corner weights is twice $|CCS|$ (minus a constant) because, when `SolveCoG` finds a new payoff vector, one corner weight is removed and three new corner weights are added. For $d > 3$, a loose bound on $|\mathcal{W}_{CCS}|$ is the total number of possible combinations of d payoff vectors or boundaries: $O(\binom{|CCS|+d}{d})$. However, we can obtain a tighter bound by observing that counting the number of corner weights given a CCS is equivalent to *vertex enumeration*, which is the dual problem of *facet enumeration*, i.e., counting the number of vertices given the corner weights (Kaibel & Pfetsch, 2003).

Theorem 10. *For arbitrary d, $|\mathcal{W}_{CCS}|$ is bounded by $O(\binom{|CCS|-\lfloor\frac{d+1}{2}\rfloor}{|CCS|-d} + \binom{|CCS|-\lfloor\frac{d+2}{2}\rfloor}{|CCS|-d})$ (Avis & Devroye, 2000).*

Proof. This result follows directly from *McMullen's upper bound theorem* for facet enumeration (Henk, Richter-Gebert, & Ziegler, 1997; McMullen, 1970). □

The same reasoning used to prove Theorems 9 can also be used to establish the following:

Corollary 1. *The runtime of VELS with $\varepsilon \geq 0$ is $O((|\varepsilon\text{-}CCS| + |\mathcal{W}_{\varepsilon\text{-}CCS}|)(n|\mathcal{A}_{max}|^w + C_{nw} + C_{heur})$, where $|\varepsilon\text{-}CCS|$ is the size of the ε-CCS, and $|\mathcal{W}_{\varepsilon\text{-}CCS}|$ is the number of corner weights of $u^*_{\varepsilon\text{-}CCS}(\mathbf{w})$.*

In practice, VELS will often not test all the corner weights of the polyhedron spanned by the ε-CCS, but this cannot be guaranteed in general. In Section 5.4, we show empirically that $|\varepsilon\text{-}CCS|$ decreases rapidly as ε increases.

12. When the reduction in Footnote 11 is used, only a very small subset of $\mathcal{W}$ is used, making it even smaller.

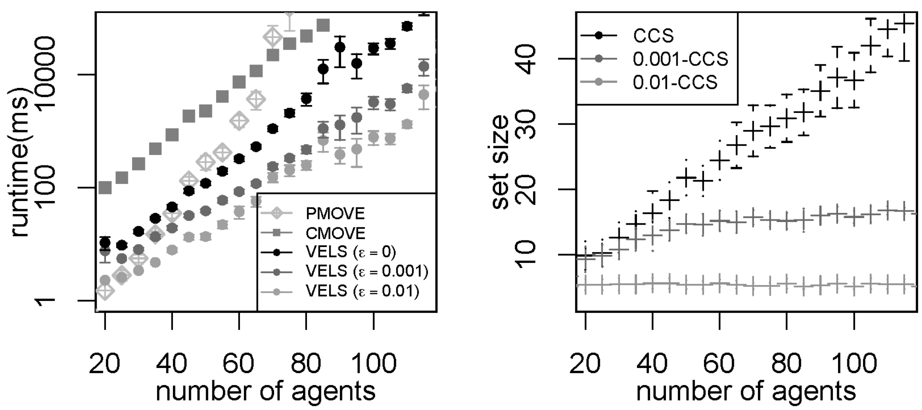


Figure 9: (left) The runtimes of PMOVE, CMOVE and VELS with different values of ε, for varying numbers of agents, n, and $\rho = 1.5n$ factors, 2 actions per agent, and 2 objectives and (right) the corresponding sizes of the ε-CCSs.

Theorem 11. *The space complexity of VELS is* $O(d|\varepsilon\text{-}CCS|+d|\mathcal{W}_{\varepsilon-CCS}|+n|\mathcal{A}_{max}|^w)$ *with* $\varepsilon \geq 0$.

Proof. OLS needs to store every corner weight (a vector of length d) in the queue, which is at most $|\mathcal{W}_{\varepsilon-CCS}|$. OLS also needs to store every vector in S (also vectors of length d). Furthermore, when `SolveCoG` is called, the memory usage of VE is added to the memory usage of the outer loop of OLS. The memory usage of VE is $n|\mathcal{A}_{max}|^w$ (Theorem 2). □

Because OLS adds few memory requirements to that of VE, VELS is almost as memory efficient as VE and thus considerably more memory efficient than CMOVE (Theorem 7).

5.4 Empirical Evaluation

We now empirically evaluate VELS, in comparison to CMOVE and PMOVE. We no longer compare against the non-graphical method as this is clearly dominated by CMOVE and PMOVE. Where we refer to CMOVE in this section, we mean basic CMOVE, as this was fastest for the tested scenarios. Like before, we use both random graphs and the Mining Day benchmark. All experiments in this section were run on a 2.4 GHx Intel Core i5 computer, with 4 GB memory.

5.4.1 Random Graphs

To test VELS on randomly generated MO-CoGs, we use the same MO-CoG generation procedure as in Section 4. To determine how the scalability of exact and approximate VELS compares to that of PMOVE and CMOVE, we tested them on random MO-CoGs with increasing numbers of agents. The average number of factors per agent was held at $\rho = 1.5n$ and the number of objectives at $d = 2$. Figure 9 shows the results, which are averaged over 30 MO-CoGs for each number of agents. Note that the runtimes on the left, on the y-axis, are in log-scale but the set sizes on the right are not.

These results demonstrate that VELS is more efficient than CMOVE for two-objective random MO-CoGs. The runtime of exact VELS ($\varepsilon = 0$) is on average 16 times less than that of CMOVE. CMOVE solves random MO-CoGs with 85 agents in 74s on average, whilst exact VELS can handle 110 agents in 71s.

While this is already a large gain, we can achieve an even lower growth rate by permitting a small ε. For 110 agents, permitting a 0.001 error margin yields a gain of more than an order of magnitude, reducing the runtime to 5.7s. Permitting a 0.01 error reduces the runtime to only 1.3s. We can thus reduce the runtime of VELS by a factor of 57, while retaining 99% accuracy. Compared to CMOVE at 85 agents, VELS with $\varepsilon = 0.01$ is 109 times faster.

These speedups can be explained by the slower growth of the ε-CCS (Figure 9 (right)). For small numbers of agents, the size of the ε-CCS grows only slightly more slowly than the size of the full CCS. However, from a certain number of agents onwards, the size of the ε-CCS grows only marginally while the size of the full CCS keeps on growing. For $\varepsilon = 0.01$, the ε-CCS grew from 2.95 payoff vectors to 5.45 payoff vectors between 5 and 20 agents, and then only marginally to 5.50 at 110 agents. By contrast, the full CCS grew from 3.00 to 9.90 vectors between 5 and 20 agents, but then keeps on growing to 44.50 at 110 agents. A similar picture holds for the 0.001-CCS, which grows rapidly from 3.00 vectors at 5 to 14.75 vectors at 50 agents, then grows slowly to 16.00 at 90 agents, and then stabilizes, to reach 16.30 vectors at 120 agents. Between 90 and 120 agents, the full CCS grows from 35.07 vectors to 45.40 vectors, making it almost 3 times as large as the 0.001-CCS and 9 times larger than the 0.01-CCS .

To test the scalability of VELS with respect to the number of objectives, we tested it on random MO-CoGs with a constant number of agents and factors $n = 25$ and $\rho = 1.5n$, but increased the number of objectives, for $\varepsilon = 0$ and $\varepsilon = 0.1$. We compare this to the scalability of CMOVE. We kept the number of agents ($n = 25$) and the number of local payoff functions ($\rho = 37$) small in order to test the limits of scalability in the number of objectives. The number of actions per agent was 2. Figure 10 (left) plots the number of objectives against the runtime (in log scale). Because the CCS grows exponentially with the number of objectives (Figure 10 (right)), the runtime of CMOVE is also exponential in the number of objectives. VELS however is linear in the number of corner weights, which is exponential in the size of the CCS, making VELS doubly exponential. Exact VELS ($\varepsilon = 0$) is faster than CMOVE for $d = 2$ and $d = 3$, and for $d = 4$ approximate VELS with $\varepsilon = 0.1$ is more than 20 times faster. However for $d = 5$ even approximate VELS with $\varepsilon = 0.1$ is slower than CMOVE.

Unlike when the number of agents grows, the size of the ε-CCS (Figure 10 (right)) does not stabilize when the number of objectives grows, as can be seen in the following table:

$\lvert\varepsilon\text{–}CCS\rvert$	$\varepsilon = 0$	$\varepsilon = 0.001$	$\varepsilon = 0.01$	$\varepsilon = 0.1$
$d = 2$	10.6	7.3	5.6	3.0
$d = 3$	68.8	64.6	41.0	34.8
$d = 4$	295.1	286.1	242.6	221.7

We therefore conclude that VELS can compute a CCS faster than CMOVE for 3 objectives or less, but that CMOVE scales better in the number of objectives. VELS however, scales better in the number of agents.

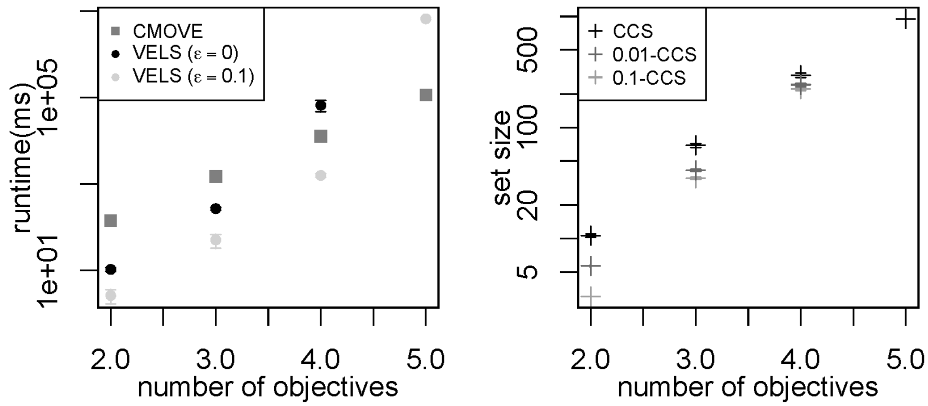


Figure 10: (left) the runtimes of CMOVE and VELS ($\varepsilon = 0$ and $\varepsilon = 0.1$), for varying numbers of objectives (right) the size of the ε-CCS for varying numbers of objectives.

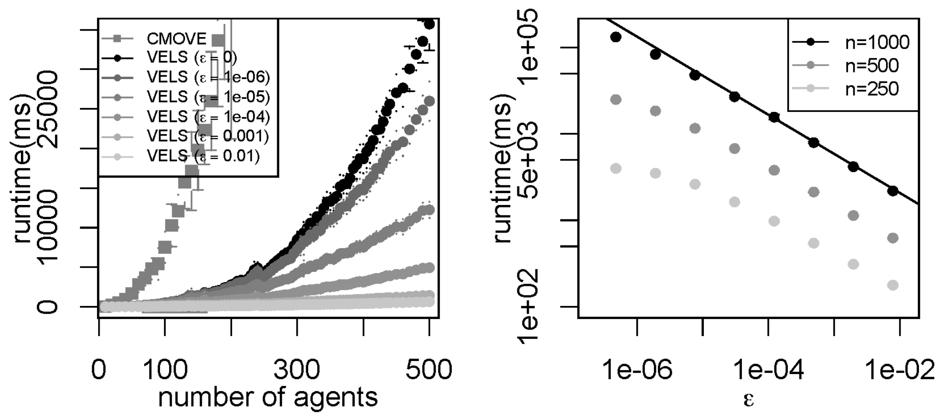


Figure 11: (left) plot of the runtimes of CMOVE and VELS with different values of ε, for varying n (up to 500). (right) loglogplot of the runtime of VELS on 250, 500, and 1000 agent mining day instances, for varying values of ε.

5.4.2 Mining Day

We now compare CMOVE and VELS on the Mining Day benchmark using the same generation procedure as in Section 4.5.2. We generated 30 Mining Day instances for increasing n and averaged the runtimes (Figure 11 (left)). At 160 agents, CMOVE has reached a runtime of $22s$. Exact VELS ($\varepsilon = 0$) can compute the complete CCS for a MO-CoG with 420 agents in the same time. This indicates that VELS greatly outperforms CMOVE on this structured 2-objective MO-CoG. Moreover, when we allow only 0.1% error ($\varepsilon = 0.001$), it takes only $1.1s$ to compute an ε-CCS for 420 agents, a speedup of over an order of magnitude.

To measure the additional speedups obtainable by further increasing ε, and to test VELS on very large problems, we generated Mining Day instances with $n \in \{250, 500, 1000\}$. We averaged over 25 instances per value of ε. On these instances, exact VELS runs in $4.2s$ for $n = 250$, $30s$ for $n = 500$ and $218s$ for $n = 1000$ on average. As expected, increasing ε leads to greater speedups (Figure 11 (right)). However, when ε is close to 0, i.e., the

ε-CCS is close to the full CCS, the speedup is small. After ε has increased beyond a certain value (dependent on n), the decline becomes steady, shown as a line in the log-log plot. If ε increases by a factor 10, the runtime decreases by about a factor 1.6.

Thus, these results show that VELS can compute an exact CCS for unprecedented numbers of agents (1000) in well-structured problems. In addition, they show that small values of ε enable large speedups, and that increasing ε leads to even bigger improvements in scalability.

6. Memory-Efficient Methods

Both CMOVE and VELS are designed to minimize the runtime required to compute a CCS. However, in some cases, the bottleneck may be memory instead. Memory-efficient methods for CoGs and related problems have recently received considerable attention (Dechter & Mateescu, 2007; Marinescu, 2008, 2009; Mateescu & Dechter, 2005). In this section, we show that, because it is an outer loop method, VELS is naturally memory efficient and can therefore solve much larger MO-CoGs than an inner loop method like CMOVE when memory is restricted. In addition, we show how both CMOVE and VELS can be modified to produce even more memory-efficient variants.

6.1 And/Or Tree Search

We begin with some background on *AND/OR tree search* (Dechter & Mateescu, 2007; Marinescu, 2008; Mateescu & Dechter, 2005; Yeoh, Felner, & Koenig, 2010), a class of algorithms for solving single-objective CoGs that can be tuned to provide better space complexity guarantees than VE. However, the improvement in space complexity comes at a price, i.e., the runtime complexity is worse (Mateescu & Dechter, 2005). The background we provide is brief; for a broader overview of AND/OR tree search for CoGs and related models please see the work of Dechter (2013) and Marinescu (2008), and for multi-objective versions the work of Marinescu (2009, 2011).

AND/OR tree search algorithms work by converting the graph to a *pseudo tree (PT)* such that each agent need only know which actions its *ancestors* and *descendants* in the PT take in order to select its own action. For example, if an agent i (a node) in the PT has two subtrees (T_1 and T_2) under it, all the agents in T_1 are conditionally independent of all the agents in T_2 given i and the ancestors of i. Figure 12a shows the PT for the coordination graph in Figure 2a.

Next, AND/OR tree search algorithms perform a tree search that results in an *AND/OR search tree (AOST)*. Each agent i in an AOST is an OR-node. Its children are AND-nodes, each corresponding to one of agent i's actions. In turn, the children of these AND-nodes are OR-nodes corresponding to agent i's children in the PT. Because each action (AND-nodes) of agent i has the same agents under it as OR-nodes, the agents and actions can appear in the tree multiple times. Figure 12b shows an AOST for the graph of Figure 2a.

A specific joint action can be constructed by traversing the tree, starting at the root and selecting one alternative from the childen of each OR-node, i.e., one action for each agent, and continuing down all children of each AND-node. For example, in Figure 12b, the joint action $< \bar{a}_1, \dot{a}_2, \dot{a}_3 >$ is indicated in grey. To retrieve the value of a joint action, we must first define the value of AND-nodes.

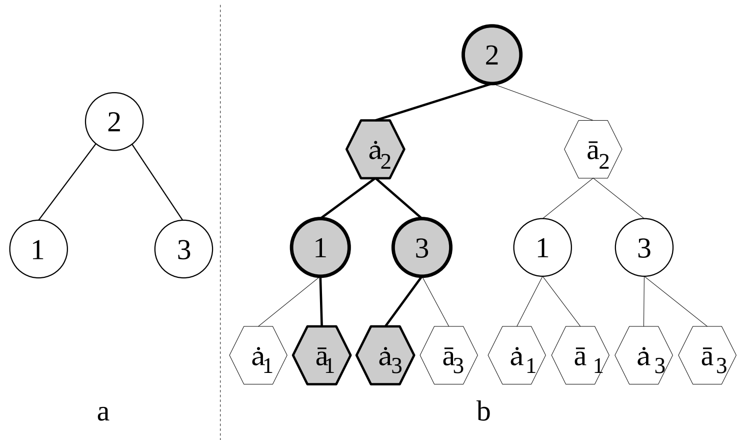


Figure 12: (a) a pseudo tree, (b) a corresponding AND/OR search tree.

Definition 18. *The value of an AND-node v_{a_i}, representing an action a_i of an agent i is the sum of the local payoff functions that have i in scope; a_i, together with its AND-node ancestors' actions, specifies an action for each agent in scope of these local payoff functions.*

For example, in Figure 12b, the total payoff of the CoG is $u(a_1, a_2, a_3) = u_1(a_1, a_2) + u_2(a_2, a_3)$. The value of the grey AND-node $\dot{a}_3$ is $u_2(\dot{a}_2, \dot{a}_3)$, as u_3 is the only payoff function that has agent 3 in scope and, together with its ancestral AND-nodes, the grey $\dot{a}_2$-node, $\dot{a}_3$ completes a joint local action for u_2.

To retrieve the optimal action, we must define the value of a subtree in the AOST:

Definition 19. *The value of a subtree $v(T_i)$ rooted by an OR-node i in an AOST is the* maximum *of the value of the subtrees rooted by the (AND-node) children of i. The value of a subtree $v(T_{a_i})$ rooted by an AND-node a_i in an AOST is the value of a_i itself (Definition 18) plus the sum of the value of the subtrees rooted by the (OR-node) children of a_i.*

The most memory-efficient way to retrieve the optimal joint action using an AOST is Euler-touring it, i.e., performing a depth-first search and computing the values of the subtrees. By generating nodes on the fly and deleting them after they are evaluated, memory usage is minimized. We refer to this algorithm simply as *AND/OR tree search* (TS). As in earlier sections, our implementation employs a tagging scheme, tagging the value of a subtree with the actions that maximize it.

While TS is a single-objective method, it has been extended to compute the PCS, yielding an algorithm we call *Pareto TS (PTS)* (Marinescu, 2009). To define PTS, we must update Definition 19 to be a set of Pareto-optimal payoffs. We refer to such a subtree value set as an *intermediate PCS (IPCS)*.

Definition 20. *The intermediate PCS of a subtree, $IPCS(T_i)$ rooted by an OR-node i is the PCS of the union of the intermediate PCSs of the children, $ch(i)$, of i:*

$$IPCS(T_i) = \texttt{PPrune}(\bigcup_{a_j \in ch(i)} IPCS(T_{a_j})).$$

The intermediate PCS of a subtree, $IPCS(T_{a_i})$ rooted by an AND-node a_i is the PCS of the value of a_i itself (Definition 18) plus the cross-sum of the intermediate PCSs of the subtrees rooted by the (OR-node) children of a_i:

$$IPCS(T_{a_i}) = \texttt{PPrune}(\left(\bigoplus_{j \in ch(a_i)} IPCS(T_j) \right) \oplus \{v_{a_i}\}).$$

Thus, PTS replaces the max operator in TS by a pruning operator, just as PMOVE replaces the max operator in VE by a pruning operator.

6.2 Memory-Efficient CCS Algorithms

We now propose two memory-efficient algorithms for computing a CCS. Both are straightforward variants of CMOVE and VELS.

The first algorithm, which we call *Convex TS (CTS)*, simply replaces `PPrune` by `CPrune` in Definition 20. Thus, CTS is like PTS but with a different pruning operator. It can also be seen as CMOVE but with VE replaced with TS. The advantage of CTS over PTS is analogous to that of CMOVE over PMOVE: it is highly beneficial to compute local CCSs instead of local PCSs because the intermediate coverage sets are input to the next subproblem in a sequential search scheme, regardless of whether that scheme is VE or TS. While CTS is more memory efficient than CMOVE, it still requires computing intermediate coverage sets that take up space. While these are typically only about as large as the CCS, their size is bounded only by the total number of joint actions.

The second algorithm addresses this problem by employing OLS with TS as the single-objective solver subroutine, `SolveCoG`, yielding *tree search linear support (TSLS)*. Thus, TSLS is like VELS but with VE replaced by TS. Because TSLS is an outer-loop method, it runs TS in sequence, requiring only the memory used by TS itself and the overhead of the outer loop, which consists only of the partial CCS (Definition 13) and the priority queue. Consequently, TSLS is even more memory efficient than CTS.

6.3 Analysis

TS has much better space complexity than VE, i.e., only linear in the number of agents n:

Theorem 12. *The time complexity of TS is $O(n|\mathcal{A}_{max}|^m)$, where n is the number of agents, $|\mathcal{A}_{max}|$ is the maximal number of actions of a single agent and m is the depth of the pseudo tree, and uses linear space, $O(n)$.*

Proof. The number of nodes in an AOST is bounded by $O(n|\mathcal{A}_{max}|^m)$. The tree creates maximally $|\mathcal{A}_{max}|$ children at each OR-node. If every AND-node had exactly one child, the number of nodes would be bounded by $O(|\mathcal{A}_{max}|^m)$, as the PT is m deep. However, if there is branching in the PT, an AND-node can have multiple children. Each branch increases the size of the AOST by at most $O(|\mathcal{A}_{max}|^m)$ nodes. Because there are exactly n agents in the PT, this can happen at most n times. At each node in the AOST, TS performs either a summation of scalars, or a maximization over scalars. Because TS performs depth-first search, at most $O(n)$ nodes need to exist at any point during execution. □

TS's memory usage is usually lower than that required to store the original (single-objective) problem in memory: $O(\rho|\mathcal{A}_{max}|^{e_{max}})$, where ρ is the number of local payoff functions in the problem, $|\mathcal{A}_{max}|$ is the maximal size of the action space of a single agent, and e_{max} is the maximal size of the scope of a single local payoff function.

The PT-depth m is a different constant than the induced width w, and is typically larger. However, m can be bounded in w.

Theorem 13. *Given a MO-CoG with induced width w, there there exists a pseudo tree for which the depth $m \leq w \log n$ (Dechter & Mateescu, 2007).*

Thus, combining Theorems 12 and 13 shows that, when there are few agents, TS can be much more memory efficient than VE with a relatively small runtime penalty.

Using the time and space complexity results for TS, we can establish the following corollaries about the time and space complexity of CTS and TSLS.

Corollary 2. *The time complexity of CTS is $O(n|\mathcal{A}_{max}|^m R)$, where R is the runtime of* `CPrune`.

Proof. $O(n|\mathcal{A}_{max}|^m)$ bounds the number of nodes in the AOST. For each node in the AOST `CPrune` is called. □

The runtime of `CPrune` in terms of the size of its input is given in Equation 3. Note that the size of the input of `CPrune` depends on the size of the intermediate CCSs of the children of a node. In the case of an AND-node, this input size is $O(|ICCS_{max}|^c)$, where c is the maximum number of children of an AND-node.[13] For OR-nodes this is $O(|\mathcal{A}_{max}||ICCS_{max}|)$.

Corollary 3. *The space complexity of CTS is $O(n|ICCS_{max}|)$, where $|ICCS_{max}|$ is the maximum size of an intermediate CCS during the execution of CTS.*

Proof. Like in TS, only $O(n)$ nodes of the AOST need to exist during any point during execution, and each node contains an intermediate CCS. □

CTS is thus much more memory efficient than CMOVE, which has a space complexity that is exponential in the induced width (Theorem 7).

Corollary 4. *The time complexity of TSLS is $O((|\varepsilon\text{-}CCS|+|\mathcal{W}_{\varepsilon\text{-}CCS}|)\ (n\ |\mathcal{A}_{max}|^m + C_{nw} + C_{heur}))$, where $m \leq w \log n$ and $\varepsilon \geq 0$.*

Proof. The proof is the same as that of Theorem 9 but with the time complexity of VE replaced by that of TS. □

In terms of memory usage, the outer loop approach (OLS) has a large advantage over the inner loop approach, because the overhead of the outer loop consists only of the partial CCS (Definition 13) and the priority queue. VELS (Theorem 11) thus has much better space complexity than CMOVE (Theorem 7). TSLS has the same advantage over CTS as VELS over CMOVE. Therefore, TSLS has very low memory usage, since it requires only the memory used by TS itself plus the overhead of the outer loop.

13. Note that c is in turn upper bounded by n but this is a very loose bound.

Corollary 5. *The space complexity of TSLS is* $O(d|\varepsilon\text{-}CCS| + d|\mathcal{W}_{\varepsilon\text{-}CCS}| + n))$*, where* $m \leq w \log n$ *and* $\varepsilon \geq 0$*.*

Proof. The proof is the same as that of Theorem 11 but with the space complexity of VE replaced by that of TS. □

As mentioned in Section 6.1, TS is the most memory-efficient member of the class of AND/OR tree search algorithms. Other members of this class offer different trade-offs between time and space complexity. It is possible to create inner loop algorithms and *corresponding* outer loop algorithms on the basis of these other algorithms. The time and space complexity analyses of these algorithms can be performed in a similar manner to Corollaries 2–5. The advantages of the outer loop methods compared to their corresponding inner loop methods will however remain the same as for TSLS and CTS. Therefore, in this article we focus on comparing the most memory-efficient inner loop method against the most memory-efficient outer loop method.

6.4 Empirical Evaluation

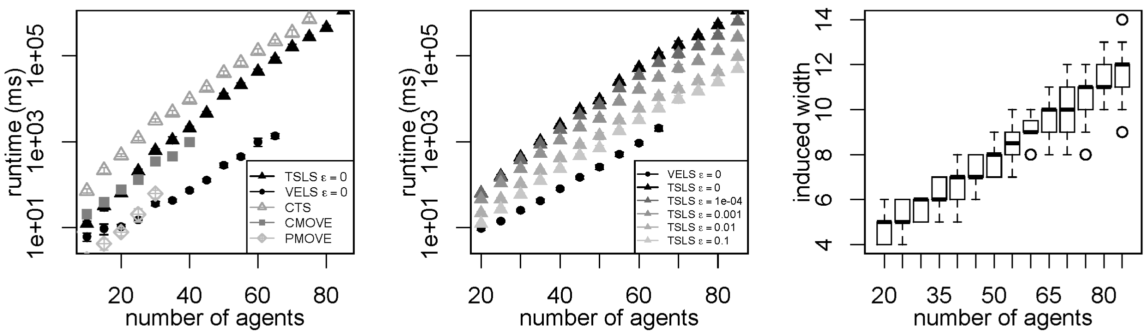
In this section, we compare CTS and TSLS to CMOVE and VELS. As before, we use both random graphs and the Mining Day benchmark. To obtain the PTs for CTS and TSLS, we use the same heuristic as CMOVE and VELS to generate an elimination order and then transform it into a PT for which $m \leq w \log n$ holds (whose existence is guaranteed by Theorem 13), using the procedure suggested by Bayardo and Miranker (1995).

6.4.1 Random Graphs

First, we test our algorithms on random graphs, employing the same generation procedure as in Section 4.5.1. Because connections between agents in these graphs are generated randomly, the induced width varies between different problems. On average, the induced width increases with the number of local payoff functions, even when the ratio between local payoff factors and the number of agents remains constant.

In order to test the sizes of problems that the different MO-CoG solution methods can handle within limited memory, we generate random graphs with two objectives, a varying number of agents n, and with $\rho = 1.5n$ local payoff functions, as in previous sections. We limited the maximal available memory to 1kB and imposed a timeout of 1800s.

Figure 13a shows that VELS can scale to more agents within the given memory constraints than the other non-memory efficient methods. In particular, PMOVE and CMOVE can handle only 30 and 40 agents, respectively, because, for a given induced width w, they must store $O(|\mathcal{A}_{max}|^w)$ local CSs. At 30 agents, the induced width (Figure 13c) is at most 6, while at 40 agents the induced width is at most 8. VELS can handle 65 agents, with an induced width of at most 11, because most of its memory demands come from running VE in the inner loop, while the outer loop adds little overhead. VE need only store one payoff in each new local payoff function that results from an agent elimination, whereas PMOVE and CMOVE must store local coverage sets. Thus, using an outer loop approach (VELS) instead of the inner loop approach (CMOVE) already yields a significant improvement in the problem sizes that can be tackled with limited memory.



(a) (b) (c)

Figure 13: (a) Runtimes in ms of TSLS, VELS, CTS, CMOVE and PMOVE on random 2-objective MO-CoGs with varying numbers of agents n and $\rho = 1.5n$ local payoff factors. (b) Runtimes of approximate TSLS for varying amounts of allowed error ε, compared to (Exact) VELS, for the same problem parameters as in (a). (c) The corresponding induced widths of the MO-CoGs in (b).

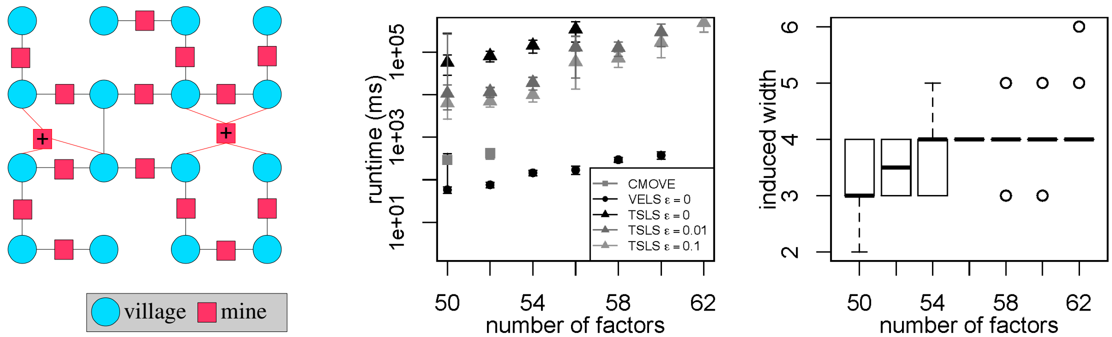
However, scaling beyond 65 agents requires a memory-efficient approach. Figure 13a also shows that, while CTS and TSLS require more runtime, they can handle more agents within the memory constraints. In fact, we were unable to generate a MO-CoG with enough agents to cause these methods to run out of memory. TSLS is faster than CTS, in this case 4.2 times faster, for the same reasons that VELS is faster than CMOVE.

However, speed is not the only advantage of the outer loop approach. When we allow a bit of error in scalarized value, ε, we can trade accuracy for runtime (Figure 13b). At 65 agents, exact TSLS ($\varepsilon = 0$), had an average runtime of $106s$, which is 51 times slower than VELS. However, for $\varepsilon = 0.0001$, the runtime was only $70s$ (33 times slower). For $\varepsilon = 0.01$ it is $11s$ (5.4 times slower), and for $\varepsilon = 0.1$ it is only $6s$ (2.9 times slower). Furthermore, the relative increase in runtime as the number of agents increases is less for higher ε. Thus, an approximate version of TSLS is a highly attractive method for cases in which both memory and runtime are limited.

6.4.2 Mining Field

We compare the performance of CMOVE and VELS against TSLS on a variation of Mining Day that we call *Mining Field*. We no longer consider CLS because it has consistently higher runtime than TSLS and worse space complexity. We use Mining Field in order to ensure an interesting problem for the memory-restricted setting. In Mining Day (see Section 4), the induced width depends only on the parameter specifying the connectivity of the villages and does not increase with the number of agents and factors. Therefore, whether or not VELS is memory-efficient enough to handle a particular instance depends primarily on this parameter and not on the number of agents.

In Mining Field, the villages are not situated along a mountain ridge but are placed on an $s \times s$ grid. The number of agents is thus $n = s^2$. We use random placement of mines, while ensuring that the graph is connected. Because the induced width of a connected grid is s and we generate grid-like graphs, larger instances have a higher induced width. The



(a) (b) (c)

Figure 14: (a) An example of a 4 by 4 Mining Field instance. The additional mines m are marked with a '+'. (b) Runtimes in ms of TSLS (for varying amounts of allowed error ε), VELS ($\varepsilon = 0$), and CMOVE on 2-objective Mining Field instances with varying numbers of additional mines $m \in [2..14]$ and a grid size of $s = 7$. (c) The corresponding induced widths of the Mining Field instances.

induced width thus no longer depends only on the connectivity parameter but also increases with the number of agents and factors in the graph.

An example Mining Field instance is provided in Figure 14a. We choose the distance between adjacent villages on the grid to be unit length. On this map, we then place the mines (local payoff functions). We connect all agents using an arbitrary tree using 2-agent local payoff functions (mines). In the figures, the mines that span this tree are unmarked and connected to the mines with black edges. We require $s^2 - 1$ factors to build the tree. Then we add m additional mines, by (independently) placing them on a random point on the map inside the grid. When a mine is placed, we connect it to the villages that are within a $r = \frac{1}{\sqrt{2}} + \eta$ radius of that mine on the map. We chose $\eta = 0.2$. Therefore, the maximum connectivity of a factor (mine) created in this fashion is 4. In the figure, these mines are marked with a '+'. The rewards per mine per worker, as well as the number of workers per village, are generated in the same way as in Mining Day.

To compare the runtimes and memory requirements of CMOVE, VELS, and TSLS on Mining Field, we tested them on a 7×7 instance (49 agents), with 1MB available memory. For TSLS, we use three different values of ε: 0 (exact), 0.01 and 0.1. We use a time limit of $1.8 \times 10^6 s$ (30 minutes). We increase the number of additional mines m from 2 (50 factors in total) onwards, by steps of 2.

Using this setup, it was not possible to solve any of the problem instances using PMOVE, which ran out of memory for all problems. In fact, PMOVE succeeded only a tree-shaped problem. i.e., one without any additional factors. Figures 14b and 14c) show the results for the remaining methods. CMOVE runs out of memory at 6 additional factors (54 factors in total). By contrast, VELS runs out of memory only at 16 additional factors, at an induced width of 6.

Compared to the random-graph results in Section 6.4.1, the induced widths of the problems that CMOVE and VELS can handle are lower in Mining Field. We suspect that this is because, on a grid-shaped problem, the number of factors with the highest induced width that need to exist in parallel during the execution of the algorithms is higher.

TSLS does not run out of memory on any of the tested instances. In face, we were unable to generate instances for which TSLS does run out of memory. However, it does run out of time. For $\varepsilon = 0$, TSLS first exceeds the time limit at $m = 10$ additional mines. For $\varepsilon = 0.01$, this happens at $m = 14$. For $\varepsilon = 0.1$, TSLS ran out of time at $m = 16$. The differences in runtime between TSLS and VELS are larger than for random graphs and therefore it is more difficult to compensate for the slower runtime of TSLS by choosing a higher ε. How much slower TSLS is compared to VELS thus seems to depend on the structure of the MO-CoG.

These Mining Field results confirm the conclusion of the random-graph experiments that using an outer loop approach (VELS) instead of the inner loop approach (CMOVE) yields a significant improvement in the problem sizes that can be tackled with limited memory. Futhermore, TSLS can be used to solve problem sizes beyond those that VELS can handle within limited memory. An approximate version of TSLS is an appealing choice for cases in which both memory and runtime are limited.

7. Conclusions and Future Work

In this article, we proposed new algorithms that exploit loose couplings to compute a CCS for multi-objective coordination graphs. We showed that exploiting these loose couplings is key to solving MO-CoGs with many agents. In particular, we showed, both theoretically and empirically, that computing a CCS has considerable advantages over computing a PCS in terms of both runtime and memory usage. Our experiments have consistently shown that the runtime of PCS methods grows a lot faster than that of our CCS methods.

CMOVE deals with multiple objectives in the inner loop, i.e., it computes local CCSs while looping over the agents. By contrast, VELS deals with multiple objectives in the outer loop, i.e., it identifies *weights* where the maximal improvement upon a partial CCS can be made and solves scalarized (single-objective) problems using these weights, yielding an anytime approach. In addition, CTS and TSLS are memory-efficient variants of these methods. We proved the correctness of these algorithms and analyzed their complexity.

CMOVE and VELS are complementary methods. CMOVE scales better in the number of objectives, while VELS scales better in the number of agents and can compute an ε-CCS, leading to large additional speedups. Furthermore, VELS is more memory-efficient than CMOVE. In fact, VELS uses little more memory than single-objective VE.

However, if memory is very restricted and VELS cannot be applied, TSLS provides a memory-efficient alternative. While TSLS is considerably slower than VELS, some of this loss can be compensated by allowing some error (ε).

There are numerous possibilities for future work. As mentioned in Section 5, OLS is a generic method that can also be applied to other multi-objective problems. In fact, (together with other authors) we already applied OLS to large multi-objective MDPs and showed that OLS can be extended to permit non-exact single-objective solvers (Roijers et al., 2014). In future work, we intend to investigate ε-approximate methods for MO-CoGs, by using ζ-approximate single-objective solvers for CoGs, using, e.g., LP-relaxation methods (Sontag, Globerson, & Jaakkola, 2011). We will attempt to find the optimal balance between the levels of approximation in the inner and outer loop, with respect to runtime guarantees and empirical runtimes.

Many methods exist for single-objective coordination graphs in which a single parameter controls the trade-off between memory usage and runtime (Furcy & Koenig, 2005; Rollón, 2008). For some of these algorithms, a corresponding multi-objective inner-loop version that computes a PCS (Marinescu, 2009, 2011) has been devised. It would be interesting to create inner and outer loop methods based on these methods that compute a CCS instead and compare performance. In particular, we have shown that OLS requires very little extra memory usage compared to single-objective solvers. It would be interesting to investigate how much extra memory could be used by a single-objective solver inside OLS, in comparison to the corresponding inner-loop method.

In addition to further work on MO-CoGs, we also aim to extend our work to sequential settings. In particular, we will look at developing an efficient planning method for multi-agent multi-objective MDPs by better exploiting loosely couplings. First, we will try to develop an ε-approximate planning version of sparse-cooperative Q-learning (Kok & Vlassis, 2006b). However, this may not be possible in general because the effects of an agent on other agents via the state is impossible to bound in general. Therefore, we hope to identify a broadly applicable subclass of multi-agent MOMDPs for which an ε-approximate planning method yields a substantial speed-up compared to exact planning methods.

Acknowledgements

We thank Rina Dechter for introducing us to memory-efficient methods for CoGs and MO-CoGs, and Radu Marinescu for his tips on memory-efficient methods and their implementation. Also, we would like to thank Maarten Inja, as well as the anonymous reviewers, for their valuable feedback. This research is supported by the NWO DTC-NCAP (#612.001.109) and NWO CATCH (#640.005.003) projects and NWO Innovational Research Incentives Scheme Veni (#639.021.336). Frans Oliehoek is affiliated with both the University of Amsterdam and the University of Liverpool.

References

Avis, D., & Devroye, L. (2000). Estimating the number of vertices of a polyhedron. *Information processing letters*, *73*(3), 137–143.

Bayardo, R. J. J., & Miranker, D. P. (1995). On the space-time trade-off in solving constraint satisfaction problems. In *IJCAI 1995: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence.*

Bertsimas, D., & Tsitsiklis, J. (1997). *Introduction to Linear Optimization.* Athena Scientific.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* Springer.

Cassandra, A., Littman, M., & Zhang, N. (1997). Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *UAI 1997: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pp. 54–61.

Cheng, H.-T. (1988). *Algorithms for partially observable Markov decision processes.* Ph.D. thesis, University of British Columbia, Vancouver.

Dechter, R. (2013). *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms*, Vol. 7 of *Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan & Claypool Publishers.

Dechter, R., & Mateescu, R. (2007). And/or search spaces for graphical models. *Artificial intelligence*, *171*(2), 73–106.

Delle Fave, F., Stranders, R., Rogers, A., & Jennings, N. (2011). Bounded decentralised coordination over multiple objectives. In *Proceedings of the Tenth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 371–378.

Dubus, J., Gonzales, C., & Perny, P. (2009). Choquet optimization using gai networks for multiagent/multicriteria decision-making. In *ADT 2009: Proceedings of the First International Conference on Algorithmic Decision Theory*, pp. 377–389.

Feng, Z., & Zilberstein, S. (2004). Region-based incremental pruning for POMDPs. In *UAI 2004: Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence*, pp. 146–153.

Furcy, D., & Koenig, S. (2005). Limited discrepancy beam search. In *IJCAI 2005: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 125–131.

Guestrin, C., Koller, D., & Parr, R. (2002). Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems 15 (NIPS'02).*

Henk, M., Richter-Gebert, J., & Ziegler, G. M. (1997). Basic properties of convex polytopes. In *Handbook of Discrete and Computational Geometry, Ch.13*, pp. 243–270. CRC Press, Boca.

Kaibel, V., & Pfetsch, M. E. (2003). Some algorithmic problems in polytope theory. In *Algebra, Geometry and Software Systems*, pp. 23–47. Springer.

Kok, J. R., & Vlassis, N. (2004). Sparse cooperative Q-learning. In *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, New York, NY, USA. ACM.

Kok, J. R., & Vlassis, N. (2006a). Using the max-plus algorithm for multiagent decision making in coordination graphs. In *RoboCup 2005: Robot Soccer World Cup IX*, pp. 1–12.

Kok, J., & Vlassis, N. (2006b). Collaborative multiagent reinforcement learning by payoff propagation. *Journal of Machine Learning Research*, *7*, 1789–1828.

Koller, D., & Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques.* MIT Press.

Lizotte, D., Bowling, M., & Murphy, S. (2010). Efficient reinforcement learning with multiple reward functions for randomized clinical trial analysis. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 695–702.

Marinescu, R., Razak, A., & Wilson, N. (2012). Multi-objective influence diagrams. In *UAI 2012: Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*.

Marinescu, R. (2008). *AND/OR Search Strategies for Combinatorial Optimization in Graphical Models*. Ph.D. thesis, University of California, Irvine.

Marinescu, R. (2009). Exploiting problem decomposition in multi-objective constraint optimization. In *Principles and Practice of Constraint Programming-CP 2009*, pp. 592–607. Springer.

Marinescu, R. (2011). Efficient approximation algorithms for multi-objective constraint optimization. In *ADT 2011: Proceedings of the Second International Conference on Algorithmic Decision Theory*, pp. 150–164. Springer.

Mateescu, R., & Dechter, R. (2005). The relationship between AND/OR search and variable elimination. In *UAI 2005: Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pp. 380–387.

McMullen, P. (1970). The maximum numbers of faces of a convex polytope. *Mathematika*, *17*(2), 179–184.

Oliehoek, F. A., Spaan, M. T. J., Dibangoye, J. S., & Amato, C. (2010). Heuristic search for identical payoff bayesian games. In *AAMAS 2010: Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1115–1122.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Pham, T. T., Brys, T., Taylor, M. E., Brys, T., Drugan, M. M., Bosman, P. A., Cock, M.-D., Lazar, C., Demarchi, L., Steenhoff, D., et al. (2013). Learning coordinated traffic light control. In *Proceedings of the Adaptive and Learning Agents workshop (at AAMAS-13)*, Vol. 10, pp. 1196–1201.

Roijers, D. M., Scharpff, J., Spaan, M. T. J., Oliehoek, F. A., de Weerdt, M., & Whiteson, S. (2014). Bounded approximations for linear multi-objective planning under uncertainty. In *ICAPS 2014: Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pp. 262–270.

Roijers, D. M., Vamplew, P., Whiteson, S., & Dazeley, R. (2013a). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, *47*, 67–113.

Roijers, D. M., Whiteson, S., & Oliehoek, F. (2013b). Computing convex coverage sets for multi-objective coordination graphs. In *ADT 2013: Proceedings of the Third International Conference on Algorithmic Decision Theory*, pp. 309–323.

Roijers, D. M., Whiteson, S., & Oliehoek, F. A. (2014). Linear support for multi-objective coordination graphs. In *AAMAS 2014: Proceedings of the Thirteenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pp. 1297–1304.

Rollón, E. (2008). *Multi-Objective Optimization for Graphical Models*. Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona.

Rollón, E., & Larrosa, J. (2006). Bucket elimination for multiobjective optimization problems. *Journal of Heuristics*, *12*, 307–328.

Rosenthal, A. (1977). Nonserial dynamic programming is optimal. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pp. 98–105. ACM.

Scharpff, J., Spaan, M. T. J., Volker, L., & De Weerdt, M. (2013). Planning under uncertainty for coordinating infrastructural maintenance. In *Proceedings of the 8th annual workshop on Multiagent Sequencial Decision Making Under Certainty*.

Sontag, D., Globerson, A., & Jaakkola, T. (2011). Introduction to dual decomposition for inference. *Optimization for Machine Learning*, *1*, 219–254.

Tesauro, G., Das, R., Chan, H., Kephart, J. O., Lefurgy, C., Levine, D. W., & Rawson, F. (2007). Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems 20 (NIPS'07)*.

Vamplew, P., Dazeley, R., Barker, E., & Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In *Advances in Artificial Intelligence*, pp. 340–349.

Yeoh, W., Felner, A., & Koenig, S. (2010). BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, *38*, 85–133.