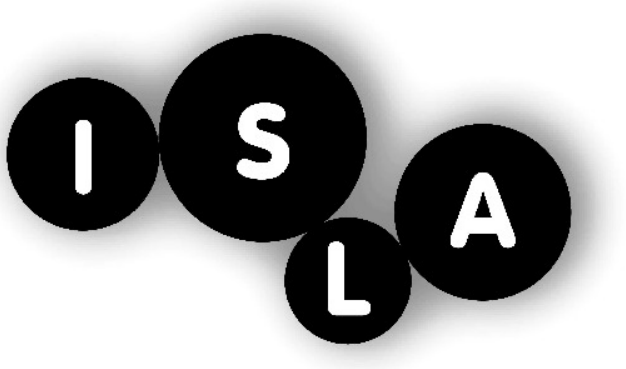
Multi-Objective Decision-Theoretic Planning

Diederik M. Roijers

ISLA Dissertation



For further information about ISLA-publications, please contact

Intelligent Systems Lab Amsterdam
Universiteit van Amsterdam
Science Park 904
1098 XH Amsterdam
phone: +31-20-525-7463
homepage: http://isla.science.uva.nl/

The investigations were supported by the DTC-NCAP (#612.001.109) project of the Netherlands Organization for Scientific Research (NWO).

Copyright © 2016 by Diederik M. Roijers
ISBN: 978–94–6328–039–6

Multi-Objective Decision-Theoretic Planning

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op dinsdag 24 mei 2016, te 10.00 uur

door

Diederik Marijn Roijers

geboren te Nijmegen

Promotor:	prof. dr. Max Welling	Universiteit van Amsterdam
Co-promotores:	dr. Shimon A. Whiteson	Universiteit van Amsterdam
	dr. Frans A. Oliehoek	Universiteit van Amsterdam
Overige leden:	prof. dr. Ann Nowé	Vrije Universiteit Brussel
	prof. dr. Karl Tuyls	University of Liverpool
	prof. dr. ir. Frans C. A. Groen	Universiteit van Amsterdam
	dr. Joris M. Mooij	Universiteit van Amsterdam
	dr. Maarten W. van Someren	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

Acknowledgments

At the end of this four-year journey, I am truly grateful to many people. I will mention several people specifically.

First, I want to thank my supervisors, Shimon and Frans, who taught me more than I ever thought I could learn. Thank you for your guidance and many interesting discussions.

I want to thank my wife, Emi, who accompanied me every step of the way. Not only did she tolerate my stress levels and long deadline nights, but actually helped me through them. Also, I want to thank my parents, Marloes and Fred, and my sister, Simone, for their constant support. This dissertation would not have been possible without you.

Thanks also to Joeri, Simone and Julienne, for proofreading and giving me feedback on my dissertation. Thanks to my co-authors (Peter, Richard, Joris, Dirk, Matthijs, Mathijs, Sander, Guangliang, Alex and Daan). It has been great working with you, and I hope we can do so again.

Special thanks to the students whom I had the pleasure of supervising and teaching (a.o., Chiel, Lieuwe, Sander, Maarten, Maarten, Auke, Camiel, Eugenio, Luisa, Timon, Philipp, Mircea, Paris, Elise, Carla, Marysia, Richard, Edwin, Maarten, and Joost).

Thanks to prof. Ann Nowé and prof. Abdel-Illah Mouaddib, for inviting me to your labs. These visits were very inspiring. Thanks to John-Jules Meyer, Linda van der Gaag, Silja Renooij, Robby Tan, Johan Jeuring and Ad Feelders, for kindling my interest in computer science and research, and inspiring me to go and do a PhD. Thanks to my colleagues and friends at the UvA (and across the street at CWI), SIKS, and the AAMAS and wider AI communities. And last but not least, thanks to all my friends and family who have supported me throughout this journey.

Leiden,
April 12, 2016.

Diederik M. Yamamoto-Roijers

Chapter 1

Introduction

A central problem in artificial intelligence is the design of artificial *autonomous agents*. An agent is "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors" (Russell et al., 1995), i.e., an artificial agent typically is a computer program — possibly embedded in specific hardware — that *takes actions* in an environment that changes as a result of these actions. An autonomous agent (Franklin and Graesser, 1997) can act autonomously, i.e., without constant human control or intervention, on a user's behalf.[1]

Artificial autonomous agents can assist us in many useful ways. For example, agents can perform the control of manufacturing machine, in order to produce products for a company (Monostori et al., 2006; Van Moergestel, 2014), drive a car instead of a human driver (Guizzo, 2011), trade goods or services on markets (Ketter et al., 2013; Pardoe, 2011) and help provide security (Tambe, 2011). As such, autonomous agents have enormous economic potential, as well as potential for improving our quality of life.

In order to perform tasks, autonomous agents require the capacity to reason about their environment and consequences of their actions — and the desirability thereof. The study of this reasoning is called *decision theory*. Decision theory uses probabilistic models of the environment. Typically these models include the *states* the environment can be in, the possible *actions* that agents can perform in each state, and how the state of the environment is affected by these actions. Furthermore, the desirability of actions and their effects are codified in numerical feedback signals. These feedback signals are typically referred to as *reward* or *payoff* functions.

Decision-theoretical models enable autonomous agents to *plan* how to act. The models encode how the environment behaves as a result of the actions agents and which observations and rewards agents can expect. Agents can use this information to formulate a *policy* that specifies agent behavior as a function of what the agents observe.

[1] For a detailed discussion on the definition of autonomous agents, see e.g., the discourses by Russell et al. (1995) and Franklin and Graesser (1997).

In most research on planning in decision problems, the desirability of actions and their effects are codified in a *scalar* reward function (Busoniu et al., 2008; Oliehoek, 2010; Thiébaux et al., 2006; Wiering and Van Otterlo, 2012). The planning task in such scenarios is to find a policy that maximizes the expected (cumulative) reward.

However, many real-world decision problems have multiple objectives. For example, for a computer network we may want to maximize performance while minimizing power consumption (Tesauro et al., 2007). In such cases the problem is more naturally expressed using a vector-valued reward function. When the reward function is vector-valued, the value of a policy is also vector-valued. Typically, there is no single policy that maximizes the value for all objectives simultaneously. For example, in the computer network example, we can achieve higher performance by using more power. Rather than producing a single optimal policy, as in single-objective planning, it may therefore be crucial to produce a set of policies that offer different trade-offs between the objectives.

In this dissertation, we focus on multi-objective decision-theoretic planning. In a multi-objective planning scenario, the agents are given a model of the environment and asked to provide a set of policies from which to elect a policy to execute. We first motivate the need for specialized multi-objective planning methods in this scenario, and introduce our perspective on what it means to solve a multi-objective decision problem. Then we introduce the research questions, the scope of this dissertation and provide an overview of the contributions in this dissertation.

1.1 Motivating Scenarios

The existence of multiple objectives in a decision problem does not automatically imply that we require specialized multi-objective methods to solve it. If the decision problem can be *scalarized*, i.e., the vector-valued reward function can be converted to a scalar reward 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* that expresses the utility of the user for different trade-offs between the objectives.

Definition 1. *A* scalarization function f*, is a function that maps a multi-objective value of a policy* π *of a decision problem,* $\mathbf{V}^{\pi}$*, to a scalar value* $V^{\pi}_{\mathbf{w}}$*:*

$$V^{\pi}_{\mathbf{w}} = f(\mathbf{V}^{\pi}, \mathbf{w}),$$

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

The second step of the conversion is to define a single-objective version of the decision problem such that the utility of each policy π equals the scalarized value of the original multi-objective decision problem $V^{\pi}_{\mathbf{w}}$.

Though it is rarely stated explicitly, all research on automated multi-objective decision making rests on the premise that there are decision problems for which one or both

of these conversion steps are impossible, infeasible, or undesirable. Here, we discuss three motivating scenarios in which this is indeed the case, thereby demonstrating the need for specialized multi-objective methods. These scenarios are depicted schematically in Figure 1.1.

Figure 1.1a provides an overview of the *unknown weights scenario*. In this scenario, $\mathbf{w}$ is unknown at the moment when planning must occur: the *planning phase*. For example, consider a company that mines different resources from different mines spread out through a mountain belt. The workers of the company live in villages at the foot of the mountains. In Figure 1.2, 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 can be mined. Different mines yield different quantities of resource per worker. The market prices per unit of resource vary through a stochastic process and every price change can change the optimal assignment of vans. Furthermore, the expected price variation increases with time. It is therefore critical to act based on the latest possible price information in order to maximize performance. Because computing the optimal van assignment takes time, redoing this computation every time the prices change is highly undesirable. Therefore, we need a multi-objective method that computes a set containing an optimal solution for every possible value of the prices, $\mathbf{w}$. We call such a set a *coverage set*, as it "covers" all possible preferences of the user (i.e., the possible values of the prices in our example) with respect to the objectives (as specified by f). Although computing a coverage set is computationally more expensive than computing a single optimal policy for a given price, it needs to be done only once. Furthermore, the *planning phase* (Figure 1.1a) can take place in advance, when more computational resources are available.

In the *selection phase*, when the prices ($\mathbf{w}$) are revealed and we want to use as little computation time as possible, we can use the coverage set to determine the best policy by simple maximization. Finally, in the *execution phase*, the selected policy is employed.

In the unknown weights scenario *a priori* scalarization is undesirable, because it would shift the burden of computation towards a point in time where it is not available. The scalarization f is known, and the weights $\mathbf{w}$ will become available in the selection phase, where a single policy is selected for execution. However, there are also settings in which $\mathbf{w}$ or even f will never be made explicit. We call this scenario the *decision support* scenario.

In the *decision support scenario* (Figure 1.1b), scalarization is infeasible throughout the entire decision-making process because of the difficulty of specifying $\mathbf{w}$ and/or f. For example, when a community is considering the construction of a new metro line, economists may not be able to accurately compute the economic benefit of reduced commuting times. The users may also have "fuzzy" preferences that defy meaningful quantification. For example, if construction of the new metro line could be made more efficient by building it in such a way that it obstructs a beautiful view, then a human designer may not be able to quantify the loss of beauty. The difficulty of specifying the exact scalarization is especially apparent when the designer is not a single person but a

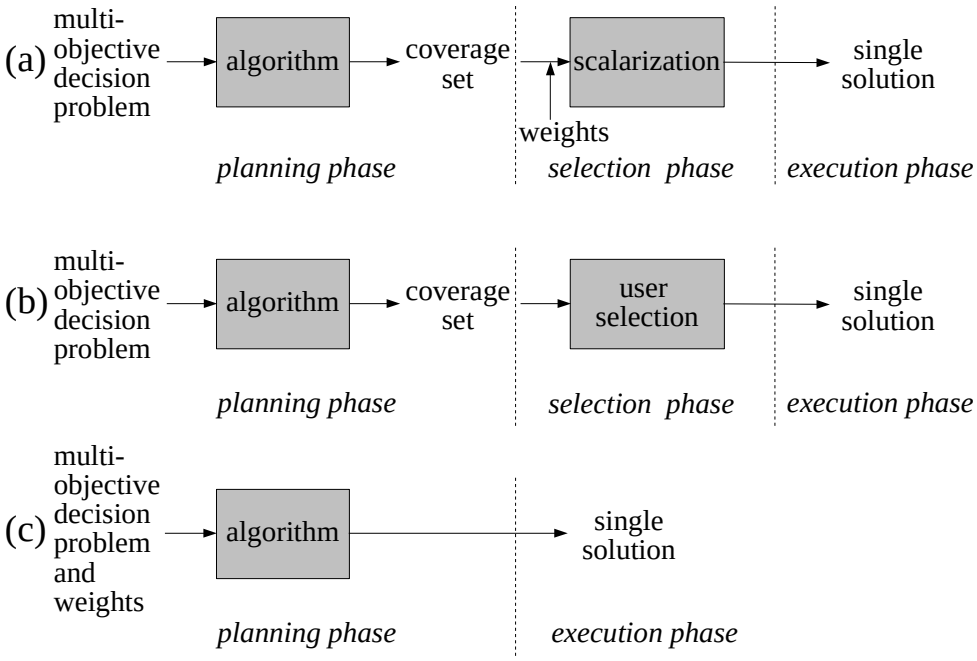


Figure 1.1: The three motivating scenarios for multi-objective decision-theoretic planning: (a) the unknown weights scenario, (b) the decision support scenario, (c) the known weights scenario.

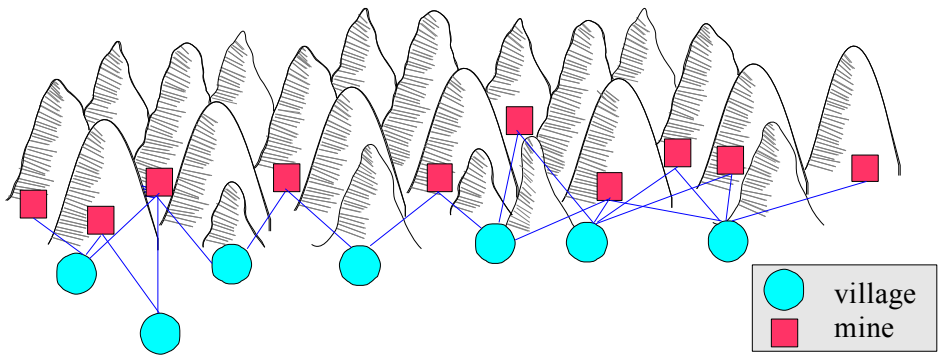


Figure 1.2: Mining company example.

committee or legislative body whose members have different preferences and agendas, such as the politicians and interest groups involved in the metro line example. In such a system, the multi-objective planning method is used to calculate a coverage set with respect to the constraints that can safely be imposed on f and $\mathbf{w}$. For example, we can safely assume that gaining value in one objective, without reducing the value in any of the others cannot reduce the utility of the user (i.e., the scalarized value).

As shown in Figure 1.1b, the decision support scenario proceeds similarly to the unknown weights scenario in the planning phase. In the selection phase however, the user or users select a policy from the coverage set according to their preferences directly, rather than explicitly computing a numerical utility by applying the scalarization function to each value vector.

In the decision support scenario, one could still argue that scalarization before planning or learning is possible in principle. For example, the loss of beauty can be quantified by measuring the resulting drop in housing prices in neighborhoods that previously enjoyed an unobstructed view. However, the difficulty with explicit scalarization is not only that doing so may be impractical but, more importantly, that it forces the users to express their preferences in a way that may be inconvenient and unnatural. This is because selecting $\mathbf{w}$ requires weighing hypothetical trade-offs, which can be much harder than choosing from a set of actual alternatives. This is a well understood phenomenon in the field of *decision analysis* (Clemen, 1997), where the standard workflow involves presenting alternatives *before* soliciting preferences. In the same way, algorithms for multi-objective decision problems can provide critical decision support; rather than forcing the users to specify f and $\mathbf{w}$ in advance, these algorithms just prune policies that would not be optimal for any f and $\mathbf{w}$ that fit the known constraints on the preferences of the users, and produce a coverage set. By producing a coverage set that contains optimal solutions across all f and $\mathbf{w}$ that fit the known constraints — rather than just all $\mathbf{w}$ for a specified f, as in the unknown weights scenario — this coverage set now offers a range of alternatives from which the users can select according to preferences whose relative importance is not easily quantified.

Finally, we present one more scenario that requires explicit multi-objective methods that we call the *known weights scenario* (Figure 1.1c). In this scenario we assume that $\mathbf{w}$ is known at the time of planning and thus scalarization would be possible. However, it may well be *undesirable* because of the difficulty of the second step in the conversion. In particular, if f is nonlinear, then the resulting single-objective problem may be much more complex than the original multi-objective problem. As a result, finding the optimal policy may be intractable whilst the original multi-objective problem is tractable. This happens for example in the case of multi-objective Markov decision processes (MOMDPs[2]), where a non-linear scalarization would lead to the loss of the additivity property on which single-objective solution methods rely (Roijers et al., 2013a).

[2]This abbreviation is also used for *mixed-observability MDPs* (Ong et al., 2010), which we do not consider in this dissertation; we use the abbreviation MOMDPs solely for multi-objective MDPs.

In contrast to the unknown weights and the decision support scenarios, in the known weights scenario, the multi-objective method only produces one policy, which is then executed, i.e., there is no separate selection phase.

The scenarios we have presented here require explicit multi-objective methods because a priori scalarization of the multi-objective decision problems, and subsequent solving with standard single-objective methods, does not apply. In this dissertation, we focus on the two multi-policy scenarios, i.e., the unknown weights and decision support scenarios, in which the goal of a multi-objective planning method is to produce a coverage set, i.e., a set that contains at least one optimal solution for each possible trade-off between the objectives (as expressed by f and $\mathbf{w}$). From this coverage set, the policy that maximizes user utility will be selected in the selection phase. The goal of the planning algorithms presented in this dissertation is to maximize user utility, by producing the best possible coverage set.

1.2 Utility-Based Approach

The goal of solving all — including multi-objective — decision problems is to maximize user utility. However, in the unknown weights and decision support scenarios, we cannot optimize this directly, because at the time when planning takes place the scalarization function, f, that maps the multi-objective values to a scalar utility, and/or its parameters, $\mathbf{w}$, are unknown. Therefore, we must compute a *coverage set* (as in Figure 1.1). This coverage set is a set of policies such that, for every possible scalarization, a maximizing policy is in the set.

In this dissertation, we argue that we should derive which policies are to be included in the coverage from what we know about f. We call this the *utility-based approach*. The utility-based approach stands in contrast to the *axiomatic approach* in which it is axiomatically assumed that the coverage set is the so-called *Pareto front*, which we define formally in Section 2.1. In short, the Pareto front is the set of all policies that are Pareto optimal. A policy is Pareto optimal when there is no other policy that has at least equal value in all objectives and has a higher value in at least one objective. Indeed, the Pareto front contains at least one optimal policy for most, if not all, scalarization functions that occur in practice. However, we argue that while the Pareto front is *sufficient* it is often not *necessary* to compute the entire Pareto front. In fact the only context in which the full Pareto front is required is for the known weights or decision-support scenarios, where the scalarization function is non-linear, and a strict conditions are imposed on the type of policies that are allowed. Therefore — as we will show — a utility-based approach often results in a much smaller coverage set, which is less computationally intensive to compute and appropriate to the needs of the user.

Another upshot of the utility-based approach is that it is possible to derive how much utility is maximally lost if it is not possible to compute an exact coverage set (Zintgraf et al., 2015). Such bounds on the loss of quality due to approximation is often crucial for a meaningful interpretation of the quality of heuristic methods, especially

when comparing algorithms (Oliehoek et al., 2015). Furthermore, the bounds provide insight for the users into the quality and reliability of the selected final policy.

1.3 Focus

This dissertation is about multi-objective decision making using autonomous agents. As such it is positioned within the field of *decision theory*. According to the Oxford dictionary, decision theory is defined as:

> The mathematical study of strategies for optimal decision-making between options involving different risks or expectations of gain or loss depending on the outcome.

Specifically, we study "strategies for optimal decision-making" given that there are multiple objectives. However, within the field of decision-making there are many decision problems "involving different risks or expectations of gain or loss depending on the outcome" which we could study. We limit the scope of our inquiry, by starting at (relatively) simple multi-objective decision problems, and adding more complicating aspects. We focus on the following aspects:

- **Single agent versus cooperative multi-agent decision problems**
  Multi-agent problems are more complex than single-agent problems, because they require coordinating the actions between the agents, and the amount of possible *joint* actions grows exponentially with the amount of agents.[3]

- **Single-shot versus sequential environments**
  In single-shot environments policies specify how to select a (joint) action for a single timestep, while in sequential settings the agents interact with the environment repeatedly, and have to consider the effect of their actions upon the future *state* of the environment.

- **Fully observable versus partially observable environments**
  In a fully observable environment the true *state* of the environment is known to the agents, while in a partially observable environment only a (possibly noisy) observation signal that correlates with the state of the environment is available to the agents.

Planning in single-agent single-shot settings is trivial, but other combinations of these aspects are not. We therefore focus on the following decision problems:

- **Multi-objective coordination graphs (MO-CoGs)**
  MO-CoGs are multi-objective multi-agent fully observable single-shot decision

[3] In this dissertation, we only consider *cooperative* multi-agent problems. We briefly discuss non-cooperative models in Section 6.2.3.

problems. MO-CoGs model the coordination problem that cooperative teams of agents face while making a single joint decision in the face of multiple objectives. The main challenge in this problem is the scalability in the number of agents, as the number of possible joint actions grows exponentially in this number.

- **Multi-objective Markov decision processes (MOMDPs)**
  MOMDPs are single-agent fully observable sequential decision problems. In this problem setting, an agent observes the state of the environment and must reason about the effects of its actions upon the environment in order to obtain a favorable expected (discounted) sum of rewards over time. The main challenge in this problem is reasoning about these effects of actions upon the environment given the possibly stochastic transitions between states of the environment.

- **Multi-objective partially observable Markov decision processes (MOPOMDPs)**
  MOPOMDPs are single-agent partially observable sequential decision problems, and differ from MOMDPs only in the aspect of observability. Partial observability significantly complicates planning, to the extent that finding an optimal solution set is typically no longer tractable, and we therefore have to settle for approximate solution sets.

For the different problem settings, we aim to find planning methods that provide a coverage set. In this dissertation, we focus on methods that are either exact, or can produce bounded approximations of the coverage set, i.e., methods that produce ε-approximate coverage sets, where ε is the maximal loss of utility for the user due to approximation. As such, heuristic multi-objective planning methods based on evolutionary algorithms (Handa, 2009a,b; Soh and Demiris, 2011a) or local search (Kooijman et al., 2015; Inja et al., 2014) are beyond the scope of this dissertation.

Another aspect that is beyond the scope of this dissertation is *learning* (Sutton and Barto, 1998; Wiering and Van Otterlo, 2012). In a learning setting, the model of the environment is unknown to the agent. Therefore, the agent must learn about its environment through interaction. However, the planning methods and learning methods are not entirely disjoint; when the agent explicitly learns a model of the environment through its interaction, it can use a planning method in order to produce a coverage set. Such *model-based* learning has been investigated extensively in single-objective settings, and has recently been introduced to multi-objective settings as well (Wiering et al., 2014). As such, the methods proposed in this dissertation can be employed as planning subroutines inside a model-based learning algorithm. Furthermore, our methods could be applied in the Bayesian reinforcement learning approach (Vlassis et al., 2012) — in which learning in an MDP can be modeled as a planning in a POMDP.

1.4 Research Questions

In this research we aim to answer the following question: *"Can we create fast multi-objective planning algorithms for cooperative decision problems that are: either single- or multi-agent, single-shot or sequential, and fully or partially observable?"*

We do so by trying to find fast multi-objective planning algorithms for the problem settings discussed in Section 1.3. Note that these problem settings do not exhaust the possible combinations of the different aspects mentioned in the research question. This is partially because some of the other combinations are either trivial or too difficult, but more importantly, because we aim to find fast multi-objective methods that are applicable to as wide a range of multi-objective decision problems as possible. Specifically, we aim to create methods that are as *modular* and as *generic* as possible.

1.5 Contributions and Outline

In this section we outline the the organization of this dissertations, and the contributions we present. Also, we indicate which of these contributions have been published before and in which papers and articles.

In this introduction (Chapter 1), we have introduced and motivated multi-objective decision-theoretic planning problems and motivated the need for specialized multi-objective planning methods by using three scenarios. Furthermore, we have introduced the utility-based approach to multi-objective decision making. The motivating scenarios and the utility-based approach were introduced in (Roijers et al., 2013a), and further discussed in (Roijers et al., 2015d; Zintgraf et al., 2015; Whiteson and Roijers, 2015).

Chapter 2 provides an extensive introduction to decision-theoretic planning in general, and multi-objective decision-theoretic planning in particular. First, we discuss what it means to solve a multi-objective decision problem, and how different assumptions about the scalarization function and the types of policy allowed lead to different coverage sets. Then we outline which decision problems are common in literature, both single-objective and multi-objective and how they relate. Finally, we make the case for a specific coverage set called a *convex coverage set (CCS)* which we use throughout this dissertation, based on the utility-based approach. We argue that it is often sufficient and less costly to compute a CCS than a *Pareto coverage set (PCS)* or *Pareto front*, which is often assumed to be the optimal solution set in literature. This chapter uses our earlier taxonomy of multi-objective decision problems (Roijers et al., 2013a).

Chapter 3 presents the *optimistic linear support (OLS)* algorithm. OLS is a generic multi-objective method that solves a multi-objective decision problem as a series of scalarized, i.e., single-objective, problems. In order to do so it repeatedly calls a single-objective subroutine that is specific to the decision problem at hand.

We refer to the approach of solving a multi-objective problem as a series of single-objective problems as the *outer loop approach*. The outer loop approach stands in contrast to the *inner loop approach*, which solves a multi-objective problems using a

series of multi-objective operations, such as solving a series of smaller multi-objective problems. A practical upshot of the outer loop approach is that any single objective algorithm can be used, when made OLS-compliant, making any improvement in the state-of-the-art for a single objective decision problem an improvement for its multi-objective counterpart.

The first version of OLS was proposed in (Roijers et al., 2014b) and (Roijers et al., 2015b), and requires an exact single-objective subroutine. This limitation was countered in (Roijers et al., 2014a) by allowing approximate single-objective solvers to be used as well. OLS was further improved in terms of both theoretic and practical runtime by allowing the reuse of values and policies found in earlier calls to the single-objective subroutine in (Roijers et al., 2015a,c).

Chapter 4 considers the multi-objective coordination graph (MO-CoG). In this chapter we propose five algorithms for MO-CoGs: two inner loop methods based on exact single-objective solvers: *convex multi-objective variable elimination (CMOVE)* and *convex AND/OR tree search (CTS)*, and two outer loop methods based on OLS that use these same exact single-objective solvers as subroutines: *variable elimination linear support (VELS)* and *AND/OR tree search linear support (TSLS)*. Finally, we propose *variational optimistic linear support (VOLS)*, an OLS-based method that uses a variational single-objective coordination graph solver called *weighted mini-buckets (WMB)* as a subroutine. Because variational methods scale much better than the exact single-objective solvers, VOLS can be used to solve much larger MO-CoGs than was previously possible. However, because WMB computes only bounded approximate solutions, so does VOLS. In VOLS we leverage the insight that the algorithm can hot-start each call to WMB by reusing the *reparameterizations* output by WMB on earlier calls, leading to additional improvements in both runtime and approximation quality.

All our proposed algorithms compute a CCS rather than a PCS, which we show to be favorable both theoretically and experimentally in many situations. We compare both the runtime and the memory complexities of the the inner loop and the outer loop methods, and compare runtimes experimentally. We indicate which methods are better for which problem settings.

The algorithms we contribute in Chapter 4 have been published earlier in (Roijers et al., 2013b,c, 2014b, 2015a,b).

Chapter 5 analyses the usage of OLS for sequential single-agent decision problems. First, we consider the fully observable setting, i.e., MOMDPs, using a problem domain with large state and action spaces called the *maintenance planning problem (MPP)*. We show how to construct multi-objective planning methods based on single-objective methods via OLS, as previously published in (Roijers et al., 2014a). Because the single-objective version of the MPP is in itself a difficult problem for which the state-of-the-art is highly problem-specific (Scharpff et al., 2013), it is beneficial to be able efficiently replace the single-objective subroutines in OLS to bring the state-of-the-art in multi-objective methods up-to-date. We run new experiments for an algorithm that combines OLS with the recent CoRe algorithm (Scharpff et al., 2016), which improved the state-of-the-art for the single-objective version of the MPP. We compare this to the

previous state-of-the-art, which was to use OLS in combination with SPUDD. Furthermore, we examine the possibility of using an approximate solver (i.e., UCT*) instead of an exact solver (as previously published in (Roijers et al., 2014a)).

Then, we shift our attention to the partially observable sequential single-agent setting, i.e., MOPOMDPs. MOPOMDPs have not been studied very much in literature, due to their high complexity. We propose the first MOPOMDP method that is reasonably scalable and produces a bounded approximation of the CCS, which we call *optimistic linear support with alpha reuse (OLSAR)*. This algorithm was previously proposed in (Roijers et al., 2015c).

Chapter 6 enumerates the main conclusions and contributions of this dissertation, discusses the implications for further work in multi-objective decision making, and identifies opportunities for future work.

Chapter 2

Background

This chapter provides background on multi-objective decision-theoretic planning for different multi-objective decision problems. First, in Section 2.1, we treat the decision problems as a black box, i.e., each policy π has an associated multi-objective value $\mathbf{V}^{\pi}$, without discussing how the policy is defined, or how it induces its value. We show what it means to *solve* a multi-objective decision problem in terms of the set of all allowed policies and how this can be derived — following the utility-based approach — from what is known about the scalarization function, f. We show that different assumptions about f lead to different solution concepts. Then, in Section 2.2, we make the decision problems more concrete by introducing a simple concrete decision problem called a *multi-objective bandit problem*, and its single-objective equivalent. Using this problem, we illustrate that in multi-objective decision problems, (dis)allowing stochastic policies can have a large impact on the attainable value, while this is typically not the case for single-objective problems. We then discuss decision problems with more structure, and discuss how they relate to each other. Finally, in Section 2.3, we provide a taxonomy of multi-objective decision problems and solution concepts based on the different assumptions about f and the set of allowed policies, Π, and make the case for a specific solution concept, called the *convex coverage set*. The convex coverage set applies to a large class of settings, has important computational advantages, and we will use this solution concept for the remainder of this dissertation.

2.1 Multiple Objectives

In this dissertation, we focus on different (cooperative) multi-objective decision problems.

Definition 2. *A cooperative* single-objective decision problem (SODP)*, consists of:*

- *a set of allowed (joint)* policies Π*,*
- *a value function that assigns a real numbered value, $V^{\pi} \in \mathbb{R}$, to each joint policy $\pi \in \Pi$, corresponding to the desirability, i.e., the utility, of the policy.*

Definition 3. *In a cooperative* multi-objective decision problem (MODP)*,* Π *is the same as in an SODP, but*

- *there are* $d \geq 2$ objectives*, and*
- *the value function assigns a* value vector*,* $\mathbf{V}^\pi \in \mathbb{R}^d$*, to each joint policy* $\pi \in \Pi$*, corresponding to the desirability of the policy* with respect to each objective.

We denote to value of policy π *in the* i*-th objective as* V_i^π*.*

Both $\mathbf{V}^\pi$ and Π may have underlying structure corresponding to the structure of the environment, which we will discuss in Section 2.2. For now, we only assume Π is known and that we can, at least in theory, compute the value of each policy.

In an SODP the value function provides a complete ordering on the joint policies, i.e., for each pair of policies π and π', V^π must be greater than, equal to, or less than $V^{\pi'}$. In contrast, in an MODP, the presence of multiple objectives means that the value function $\mathbf{V}^\pi$ is a vector rather than a scalar. Such value functions supply only a partial ordering. For example, it is possible that, $V_i^\pi > V_i^{\pi'}$ but $V_j^\pi < V_j^{\pi'}$. Consequently, unlike in an SODP, we can no longer determine which values are optimal without additional information about how to prioritize the objectives, i.e., about what the *utility* of the user is for different trade-offs between the objectives.

In the *unknown weights* and *decision support scenarios* (Figure 1.1), the parameters of the scalarization function $\mathbf{w}$, or even f itself, are unknown during the *planning phase*. Therefore, in order to optimize the utility for the user, the agent has to provide a solution set. Given a solution set, the user can then pick the policy that maximizes his utility in the *selection phase*.

We want the solution set to contain at least one optimal policy for every possible scalarization (in order to guarantee optimality), but we also want the solution set to be as small as possible, in order to make the selection phase as efficient as possible.

In this dissertation, we advocate the *utility-based approach* (Roijers et al., 2013a) for determining which policies the solution set should contain. The utility-based approach rests on the following premise: before the execution phases of the scenarios of Figure 1.1, one policy is selected by collapsing the value vector of a policy to a scalar utility, using the scalarization function (Definition 1). The application of the scalarization function may be implicit or hidden, e.g., it may be embedded in the thought-process of the user, but it nonetheless occurs. The scalarization function is thus an integral part of the notion of utility, i.e., what the agent should maximize. Therefore, if we find a set with an optimal solution for each possible weight setting of the scalarization function, we have solved the MODP.

The utility-based approach stands in contrast to the *axiomatic approach* to optimality in multi-objective decision problems that is followed in a lot of multi-objective research. The axiomatic approach begins with the axiom that the optimal solution set is the Pareto front (which we define later in this section). This approach is limiting because, as we demonstrate in Section 2.3, there are many settings for which other solution sets are more suitable.

2.1.1 Undominated Sets

We will now derive the appropriate solution sets, as a subset of Π, for different assumptions about f and $\mathbf{w}$. The first thing we do is remove all policies that can never be optimal for any allowed choice of f and $\mathbf{w}$. Such policies are called *dominated*. For a dominated policy, for every choice of f and $\mathbf{w}$ within the provided constraints, there is some other policy in Π that has a higher scalarized value. When we remove all dominated policies from Π, we call the resulting set the *undominated set (U)*. When we refer to the set all allowed scalarization functions as $\mathcal{F}$, i.e., the family of permitted scalarization functions, we can define U as follows.

Definition 4. *The* undominated set (U) *of an MODP, is the set of all policies and associated value vectors that are optimal for some* $\mathbf{w}$ *of a scalarization function* $f \in \mathcal{F}$.

$$U(\Pi) = \left\{ \mathbf{V}^{\pi} : \pi \in \Pi \ \wedge \ \exists f \in \mathcal{F} \ \exists \mathbf{w} \ \forall \pi' \in \Pi \ \ f(\mathbf{V}^{\pi}, \mathbf{w}) \geq f(\mathbf{V}^{\pi'}, \mathbf{w}) \right\}.$$

For convenience, we assume that payoff vectors in $U(\Pi)$ contain both the value vectors and associated policies.

A minimal assumption is that f is monotonically increasing, i.e., if the value for one objective V_i^{π}, increases while all $V_{j \neq i}^{\pi}$ stay constant, the scalarized value $V_{\mathbf{w}}^{\pi}$ cannot decrease. This assumption ensures that objectives are desirable, i.e., all else being equal, having more of them is always better. When $\mathcal{F}_{MI}$ is the set of strictly monotonically increasing scalarization functions, the undominated set is called the *Pareto front*.

Definition 5. *The* Pareto front *is the undominated set for arbitrary strictly monotonically increasing scalarization functions,* $\mathcal{F}_{MI}$.

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

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

Computing P-dominance requires only pairwise comparison of value vectors (Feng and Zilberstein, 2004).[1]

A highly prevalent case is that in addition to f being monotonically increasing, we also know that it is linear, i.e., the parameter vectors $\mathbf{w}$ are weights by which the values of the individual objectives are multiplied.

Definition 6. *The (monotonically increasing)* linear scalarization function *is a weighted sum of the objectives, for a weight vector* $\mathbf{w}$.

$$f = \mathbf{w} \cdot \mathbf{V}^{\pi}$$

In the context of linear scalarization functions, we denote the weight for objective i *as* w_i*. Because* f *is monotonically increasing* $\forall i \ \ w_i \geq 0$.

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

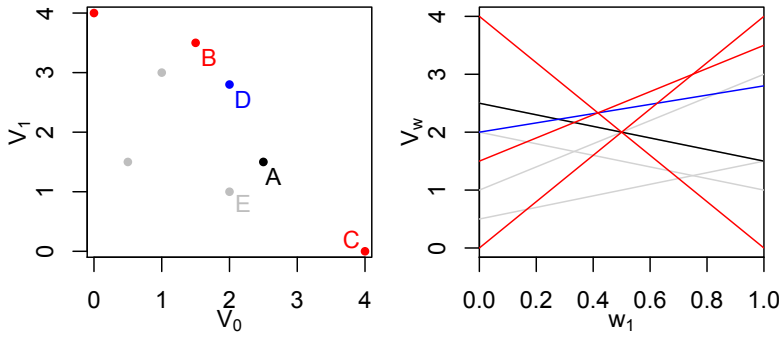


Figure 2.1: The CH and CCS versus the PF / PCS.

In the mining example from Figure 1.2, f is linear: 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 et al., 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, the undominated set is the *convex hull (CH)*[2]:

Definition 7. *The* convex hull (CH) *is the undominated set for non-decreasing linear scalarizations* $f(\mathbf{V}^\pi, \mathbf{w}) = \mathbf{w} \cdot \mathbf{V}^\pi$*:*

$$CH(\Pi) = \left\{ \mathbf{V}^\pi : \pi \in \Pi \ \wedge \ \exists \mathbf{w} \forall \pi' \quad \mathbf{w} \cdot \mathbf{V}^\pi \geq \mathbf{w} \cdot \mathbf{V}^{\pi'} \right\},$$

where $\mathbf{w}$ *adheres to the simplex constraints, i.e.,* $\forall i \ \ w_i \geq 0$ *and* $\sum_i w_i = 1$.

That is, the CH contains all solutions that attain the optimal value for at least one $\mathbf{w}$.

Vectors not in the CH are called $C\text{-}dominated$. In contrast to P-domination, C-domination cannot be tested by using pairwise comparisons because it can take two or more value vectors to C-dominate a value vector, $\mathbf{V}^\pi$. The difference between the CH and the PF is illustrated in Figure 2.1. On the left the values of all policies in Π of a 2-objective MODP are represented as points in value-space. The red (such as B and C) and blue (D) points are both in the PF and in the CH. The black point (A) is in the PF, but not in the CH. The gray points are nor in the PF nor in the CH, e.g., E is not in the PF/CH because it is P-dominated by A. On the right, the scalarized values, $V_\mathbf{w} = \mathbf{w} \cdot \mathbf{V}$, of the policies in Π are shown as a function of $\mathbf{w}$ of the linear scalarization function. Each line on the right corresponds to a point on the left. For example, the black line corresponds to the scalarized value for A as a function of $\mathbf{w}$ and the blue

[2]Note that the term *convex hull* is overloaded. In geometry (e.g., (Jarvis, 1973)), the convex hull is a superset of what we mean by the convex hull in this dissertation.

line corresponds to D. Note that because of the simplex constraints $w_2 = 1 - w_1$. C-domination of a policy π means that there is no $\mathbf{w}$ such that $V^{\pi}_{\mathbf{w}} = \max_{\pi' \in \Pi} \mathbf{w} \cdot \mathbf{V}^{\pi'}$. Note that this is true for the black line (corresponding to point A; the point that is in the PF but not the CH): even though there is no *single* other policy that is better *for every* $\mathbf{w}$, there is always some policy for every $\mathbf{w}$. For D, represented by the blue line, we observe that it is in the CH, because there is exactly one $\mathbf{w}$ for which it is optimal. However, note that at that $\mathbf{w}$ there are two other policies that achieve the same scalarized value (i.e., B and C).

2.1.2 Coverage Sets

The undominated set, $U(\Pi)$, contains all policies that are optimal for some $f \in \mathcal{F}$ and a parameterization, $\mathbf{w}$, thereof. Although this set contains no policies that are dominated, it may well contain *redundant* policies. In fact, we only need a set with at least *one* optimal policy for every f and $\mathbf{w}$. We call such a *lossless* subset of U a *coverage set*, as it covers every f and $\mathbf{w}$ with an optimal policy.

Definition 8. *A* coverage set (CS)*,* $CS(\Pi)$*, is a subset of* U*, such that for each possible* $\mathbf{w}$*, there is at least one optimal solution in the CS, i.e.,*

$$\forall f \in \mathcal{F}\ \forall \mathbf{w}\ \exists \pi \quad \Big(\mathbf{V}^{\pi} \in CS(\Pi) \wedge (\forall \pi' \quad f(\mathbf{V}^{\pi}, \mathbf{w}) \geq f(\mathbf{V}^{\pi'}, \mathbf{w}) \ ) \Big) .$$

Note that a 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 value vectors and associated policies.

For arbitrary monotonically increasing scalarization functions, we call the CS a *Pareto coverage set (PCS)*. Due to the minimal constraints on f however, we can only remove policies that have the exact same value as another policy.

Definition 9. *A* Pareto coverage set (PCS)*,* $PCS(\Pi) \subseteq PF(\Pi)$*, is a lossless subset of* $PF(\Pi)$*, i.e., it only needs to contain each unique value-vector in the PF once:*

$$\mathbf{V}^{\pi} = \mathbf{V}^{\pi'} \rightarrow \Big(\mathbf{V}^{\pi} \in PCS(\Pi) \vee \mathbf{V}^{\pi'} \in PCS(\Pi) \vee \mathbf{V}^{\pi} \notin PF(\Pi) \Big) .$$

Note that the PF itself is a PCS, but that there may be smaller PCSs.

A lossless subset of the CH with respect to linear scalarizations is called a *convex coverage set (CCS)*. That is, a CCS retains at least one policy from the CH that maximizes the scalarized payoff, $\mathbf{w} \cdot \mathbf{V}^{\pi}$, for every $\mathbf{w}$:

Definition 10. *A* convex coverage set (CCS)*,* $CCS(\Pi) \subseteq CH(\Pi)$*, is a CS for linear non-decreasing scalarizations, i.e.,*

$$\forall \mathbf{w} \exists \pi \quad \Big(\mathbf{V}^{\pi} \in CCS(\Pi) \wedge \forall \pi' \ \mathbf{w} \cdot \mathbf{V}^{\pi} \geq \mathbf{w} \cdot \mathbf{V}^{\pi'} \Big) .$$

Because 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.

Because we aim to optimize the utility for the user, we have solved an MODP once we have found a coverage set, as it contains at least one optimal policy for each f and $\mathbf{w}$, and in the selection phase, the user cannot lose utility by having a $CS(\Pi)$ instead of $U(\Pi)$. For example, for linear scalarizations and the example MODP of Figure 2.1, we do require all the policies shown in red, but we do not require the blue policy, because for every $\mathbf{w}$ there is a red policy with at least equal scalarized value.

2.1.3 Approximate Coverage Sets

A coverage set constitutes an optimal solution with respect to user utility. However, in practice it might not always be feasible to compute an exact PCS or CCS. For example, there just might not be enough runtime to compute it, or it might be too large to deal with during selection. In such cases we need to consider approximate versions of these coverage sets. Following the utility based approach, we have to limit the loss of user utility as much as possible. In order words, we focus on the *maximum utility loss (MUL)* with respect to f and $\mathbf{w}$, which are not known exactly in the planning phase.

We assume that we only know that f is monotonically increasing in all objectives (leading to an approximate PCS), or that we also know f to be linear (leading to an approximate CCS.[3] A given approximate solution set, S, should thus contain a policy, for every f and $\mathbf{w}$, for which the MUL is at most a constant.

Definition 11. *For a given solution set S and some family of scalarization functions $\mathcal{F}$, the maximum utility loss $MUL(S, \mathcal{F})$ is the maximum scalarized value that is lost due to approximation:*

$$\forall f \in \mathcal{F} \;\; \forall \mathbf{w} \;\; \forall \mathbf{V}^{\pi} \in CS(\Pi) \;\; \exists \mathbf{V}^{\pi'} \in S \quad f(\mathbf{V}^{\pi}, \mathbf{w}) \leq f(\mathbf{V}^{\pi'}, \mathbf{w}) + MUL(S, \mathcal{F}),$$

where $CS(\Pi)$ is the coverage set appropriate w.r.t. $\mathcal{F}$.

Several approximate versions of PCSs have been proposed. One of the most popular is the ε-PCS (Zitzler et al., 2003).[4] There are multiple definitions of the ε-PCS; here, we provide the definition of the so-called additive ε-PCS.[5]

Definition 12. *A given solution set S is an ε-PCS if*

$$\forall \mathbf{V}^{\pi} \in PCS(\Pi) \;\; \exists \mathbf{V}^{\pi'} \in S \;\; : \;\; \forall i = 1, \ldots, d : \; V_i^{\pi} \leq V_i^{\pi'} + \varepsilon,$$

where d is the number of objectives.

[3] For discussions about what happens if other prior information is available, please refer to (Roijers et al., 2014a) and (Zintgraf et al., 2015).

[4] The ε-PCS is called ε-approximate Pareto front in (Zitzler et al., 2003). We use ε-PCS for consistency with the terminology in this dissertation.

[5] Besides an additive ε-PCS there is also a multiplicative ε-PCS. Please refer to (Zintgraf et al., 2015) for details.

Note that an ε-PCS may in fact not contain any undominated solutions — $S \cap PF(\Pi)$ may be an empty set — but at least the maximal difference between a value vector in the PCS and the closest value vector in the ε-PCS is at most ε in all dimensions. However, when we look at the MUL of an ε-PCS for arbitrary monotonically increasing scalarizations, we immediately notice a problem: *any* increase in any objective may lead to an infinite increase in user utility. Therefore, it is impossible to compute the MUL of an ε-PCS, without more information about f and $\mathbf{w}$. For example, if we know that f is Lipschitz-continuous with a Lipschitz-constant L, the MUL is bounded by $\varepsilon\sqrt{d}L$ (Zintgraf et al., 2015).

For linear scalarization functions, we have much more information, i.e., we know the exact shape of f, and that $\mathbf{w}$ adheres to the simplex constraints. In this case it is possible to formulate an ε-CCS where ε is the MUL.

Definition 13. *A given solution set S is an ε-CCS if*

$$\forall \mathbf{w} \quad \max_{\mathbf{V}^{\pi} \in CCS(\Pi)} \mathbf{w} \cdot V_i^{\pi} - \max_{\mathbf{V}^{\pi'} \in S} \mathbf{w} \cdot V_i^{\pi'} \leq \varepsilon,$$

where $\mathbf{w}$ is a linear weight vector adhering to the simplex constraints.[6]

Note that an ε-PCS is automatically an ε-CCS, though most probably not a minimally sized one.

In Chapter 3 we propose a bounded approximate solution method for computing CCSs in MODPs, i.e., methods that come with the guarantee that they can produce an ε-CCS, for any value of ε, within finite time. Typically, the closer ε is set to 0, the longer the algorithms take to terminate.

2.2 Overview of Concrete Decision Problems

Now that we have derived the solution to MODPs, as well as bounded approximations thereof, we move to concrete MODPs to solve. In Section 1.3 we limited our scope to cooperative multi-objective decision problems. Furthermore, we discussed three aspects of decision problems: single- or multi-agent, single-shot or sequential, and fully or partially observable. Note that the first option is always the more restrictive: the most restrictive model would thus be a single-agent, single-shot, fully observable MODP.

In this section, we discuss the different models that result from different combinations of the three aspects, and place the models for which we propose new methods in context. Before doing so however, we first treat a single multi-objective decision problem with very little structure that illustrates some fundamental differences between solving single-objective and multi-objective decision problems. Specifically, we go

[6] $\mathbf{w}$ can always be made to adhere to the simplex constraints by dividing with a constant c. If $\mathbf{w}$ is not on the simplex, the MUL reported here should be multiplied by this c.

into how for cooperative SODPs, restricting the set of allowed policies Π by disallowing stochasticity typically does not affect the optimal attainable utility, while in MODPs the optimal utility is affected.

2.2.1 Bandit Problems

The simplest SODP is the multi-armed bandit problem (BP) (Sutton and Barto, 1998). BPs have very little structure to exploit, and therefore planning is either trivial — when the problem is small — or intractable — when the problem is too large. However, it is a useful problem in order to illustrate the basic concepts of decision problems.

In a BP, an agent can select an action a from a discrete set of possible actions $\mathcal{A}$. The environment provides a reward (possibly stochastically) on the basis of this action, i.e., each action has an associated expected reward $R(a)$.

Definition 14. *A multi-armed bandit problem (BP) is a tuple $\langle \mathcal{A}, R \rangle$, where*

- *$\mathcal{A}$ is a discrete set of* actions*, also called* arms*, and*
- *R is the reward function, that specifies an expected reward $R(a) \in \mathbb{R}$ for each action.*

A *policy*, π, for a BP is a probability distribution over actions, $\mathcal{A} \to [0, 1]$. The *value* of a policy π, V^π, is the expected reward of the policy:

$$V^\pi = \sum_{a \in \mathcal{A}} \pi(a) R(a).$$

A special case of a policy is the *deterministic* policy, in which one action will be chosen with probability 1. In other words, a deterministic policy in a BP is a single action. Policies that are not deterministic are called *stochastic*.

Planning in a single-objective BP is straight-forward. In the planning setting we know the model, and therefore we can chose an optimal deterministic policy by simple maximization. There is always a deterministic policy that is optimal, because there is always an action a that maximizes the reward, and choosing a different action a' cannot improve the value. This does not imply that it is always possible to retrieve the optimal policy in practice though; maximization can be infeasible when the number of actions is too large. However, because BPs do not have any *structure* that can be exploited to compute the optimal policy more efficiently, there is no way to mitigate that by clever algorithms.

In the multi-objective case, i.e., a *multi-objective multi-armed bandit problem (MOBP)* (Drugan and Nowé, 2013), we typically need more than one policy, and deterministic policies no longer suffice.

Definition 15. *A multi-objective multi-armed bandit problem (MOBP) is a tuple $\langle \mathcal{A}, \mathbf{R} \rangle$, where*

- $\mathcal{A}$ *is a discrete set of* actions*, and*
- $\mathbf{R}$ *is the reward function, that specifies an expected reward* $\mathbf{R}(a) \in \mathbb{R}^d$ *for each action, where* d *is the number of objectives.*

In the MOBP case the value of a policy, $\mathbf{V}^\pi = \sum_{a \in \mathcal{A}} \pi(a)\mathbf{R}(a)$, is vector-valued. This means that there is no longer a single action that maximizes the immediate reward.

We now illustrate which policies are required for the PCS and CCS, using an example MOBP. Imagine a 2-objective MOBP with three actions, a_1, a_2 and a_3, for which the corresponding rewards are:

- $\mathbf{R}(a_1) = (3, 0)$,
- $\mathbf{R}(a_2) = (1, 1)$, and
- $\mathbf{R}(a_3) = (0, 3)$.

When we only allow deterministic policies (of which there are three), all policies are Pareto optimal and in the PCS. However, when we determine the CCS, we see that a_2 is C-dominated, because there is no linear weight $\mathbf{w}$ for which $\mathbf{w} \cdot (1, 1)$ is better than both $\mathbf{w} \cdot (3, 0)$ and $\mathbf{w} \cdot (0, 3)$.

Now, let us allow stochastic policies. First, we observe that a stochastic policy can P-dominate a deterministic policy. The deterministic policy of always performing action a_2 is dominated by the stochastic policy $\pi(a_1) = \pi(a_3) = 0.5$ with value $(1.5, 1.5)$. In fact, we can see that all policies for which $\pi(a_2) = 0$ are Pareto optimal, and a PCS necessarily consists of all of these policies. For the CCS we also see that all policies for which $\pi(a_2) = 0$ are C-undominated. However, all but two of these policies — the deterministic policies π_1, always selecting action a_1, and π_3 always selecting a_3 — are only optimal for the weight $\mathbf{w} = (0.5, 0.5)$ and π_1 and π_3 also maximize the scalarized value for this $\mathbf{w}$. Therefore, a minimally sized CCS would still only consist these two deterministic policies. We therefore make the following observations:

Observation 1. *The PCS of deterministic policies can contain policies that are dominated by policies in the PCS of stochastic policies. The PCS of stochastic policies can be infinitely large.*

Observation 2. *The CH of stochastic policies can be infinitely large, but a CCS can still consist of a discrete set of policies.*

Let us formalize the intuition of Observation 2. We can in fact show that for computing a CCS, even when stochastic policies are allowed, we can restrict ourselves to only deterministic policies.

Theorem 1. *For an MODP that can be expressed as an MOBP, there is always a CCS with only deterministic policies, even when stochastic policies are allowed.*

Proof. We observe that for all weights $\mathbf{w}$ in a linear scalarization function, we can translate the MOBP to a single-objective BP, by redefining the reward function as the inner product of $\mathbf{w}$ with the multi-objective reward function: for all a the reward becomes $R_{\mathbf{w}}(a) = \mathbf{w} \cdot \mathbf{R}(a)$. For the resulting BP we know that there exists an optimal deterministic policy. □

This is an important result, because it guarantees that we can define a finite-size CCS. Furthermore, we generalize this proof for more complex planning problems in the following chapters, when we introduce each specific MODP.

For the stochastic PCS, a theorem similar to Theorem 1 does not hold, and the PCS can be infinitely large. However, in Section 2.3 we argue that we can mitigate this by using a compact representation of a stochastic PCS using a deterministic CCS. For now, we assume that solving an MOBP consists of either computing a deterministic PCS, or a deterministic CCS. This can be done by putting all the values of the deterministic policies in a set and subsequently removing all policies that are dominated. We refer to the removal of dominated policies from a set as *pruning* (Feng and Zilberstein, 2004).

Planning in MOBPs is nothing more than pruning away all P-dominated or C-dominated actions. The challenge in both BPs and MOBPs arises when the reward function $\mathbf{R}$ is unknown to the agent, and information about these rewards can only be attained through repeated interaction. This *learning setting* (Sutton and Barto, 1998; Wiering and Van Otterlo, 2012) poses an interesting challenge because the agent should balance exploring its options to learn more about $\mathbf{R}$ and exploiting what it already knows in order to attain high rewards. However, the learning setting is beyond the scope of this dissertation. Please refer to (Auer and Ortner, 2010; Kuleshov and Precup, 2014) for an overview of BP learning algorithms, and to (Drugan and Nowé, 2013; Yahyaa et al., 2014) for MOBP algorithms.

MOBPs are problems with very little structure, making it an uninteresting problem for planning; it is either possible to compute a CS by pruning or it is not, and it the latter case, nothing can be done about it. Therefore, we focus attention on MODPs with a more structure, that can be exploited algorithmically. In fact, the structured MODPs that we treat in this dissertation can be reduced to MOBPs, by discarding all structure. However, doing so typically makes these problems intractable.

2.2.2 Overview of Decision Problems

The MOBP is a very simple model, that is well-suited to model single-agent, single-shot and fully observable decision problems. When we relax either the first or the second of these constraints though, the planning problem will be more structured, and planning becomes more complex.

An overview of SODPs, as a Venn diagram, is presented in Figure 2.2. The different models in this diagram are: the *(multi-armed) bandit problem (BP)*, the *coordination graph (CoG)*, the *Markov decision process (MDP)*, the *multi-agent Markov decision process (MMDP)*, the *partially observable Markov decision process (POMDP)*, and

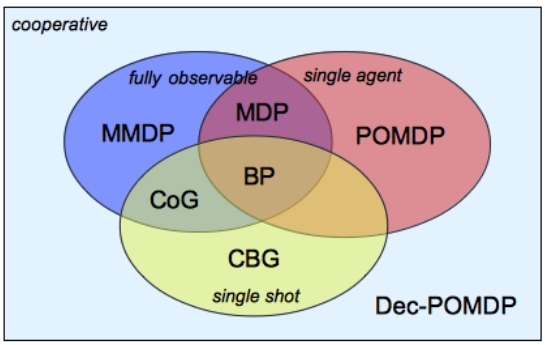


Figure 2.2: Venn diagram of cooperative (single-objective) decision problems

finally the *decentralized partially observable Markov decision process (Dec-POMDP)*. All these models have multi-objective counter-parts, that can be defined by replacing the scalar reward function in these models by a vector-valued one.

The Models in this Dissertation

In this dissertation, we limit our scope to three specific MODPs models: *multi-objective coordination graphs (MO-CoGs)*, and *multi-objective Markov decision processes (MOMDPs)* and *multi-objective partially observable Markov decision processes (MOPOMDPs)*.

MO-CoGs are cooperative single-shot, fully observable, multi-agent decision problems. In MO-CoGs, agents must coordinate their behavior in order to find effective policies. Key to making coordination between agents efficient is exploiting *loose couplings*, i.e., each agent's actions directly affect only a subset of the other agents. Such loose couplings are expressed by a reward function, that decomposes into a sum over (many) local reward functions in which only subsets of the agents participate. We define the MO-CoG model formally in Chapter 4.

As we discuss in Chapter 4, it is possible to flatten a MO-CoG to an MOBP, by ignoring the graphical structure of the reward function. Such a flattening can be seen as defining a single central control agent that has the Cartesian product of the individual action spaces of all agents as its action space. Because the size of this Cartesian product grows exponentially with the number of agents tin the problem however, this approach is typically intractable. It is however important to note that because this flattening is possible, Theorem 1 applies.

MOMDPs are single-agent, fully observable, sequential decision problems. A policy in an MOMDP thus consists of a sequence of (probability distributions over) actions. This sequence executed actions that results from a policy affect the environment. Therefore, the agents do not only have to consider their immediate reward, but also the reward they will be able attain later, by changing the state of the environment to a more favorable one. Because the effects of the actions are typically stochastic, finding suitable policies for defining a coverage set requires reasoning over all possible

future states of the environment. We define the MOMDP model formally in Chapter 5. While it is not straight-forward to flatten an MOMDP to an MOBP, it is known that we can define a CCS consisting of deterministic policies even when stochastic policies are allowed.

Also in Chapter 5, we discuss MOPOMDPs, which are single-agent, partially observable, sequential decision problems. While this partial observability poses an important additional challenge, it important to note that a reduction exists to a multi-objective MDP, be it with a continuous state-space. Therefore, is possible to define a CCS consisting of deterministic policies even when stochastic policies are allowed, with respect to this continuous state. We discuss how this works in detail in Section 5.1.2.

Other Models

For the MO-CoG, MOMDP and MOPOMDP models we propose new algorithms in the following chapters. However, there are more possible collaborative MODPs, that extend SODPs from Figure 2.2. In particular, these models represent other combinations of the aspects we discussed in Section 1.3:

- (Multi-objective) *collaborative bayesian games (CBGs)* (Oliehoek et al., 2012) are multi-agent, single-shot, and partially observable decision problems.
- (Multi-objective) *multi-agent Markov decision processes (MMDPs)* (Boutilier, 1996) are multi-agent, sequential, and fully observable decision problems.
- (Multi-objective) *decentralized partially observable Markov decision process (Dec-POMDP)* (Bernstein et al., 2002; Oliehoek, 2010) are multi-agent, sequential, and partially observable decision problems.

We do not propose new methods for these models in this dissertation, but we do discuss the implications of our work for these models in Section 6.2.3 of Future Work.

2.3 Case for the Convex Coverage Set

Now that we have introduced different solution concepts for multi-objective decision problems (MODPs) as well as specific instances of MODPs, we advocate a specific solution concept, i.e., the convex coverage set (CCS) of deterministic policies. As explained in Section 2.1.2, the CCS is an exact solution when the scalarization function, f (Definition 1), is linear. In this section, we show that the CCS is also a sufficient set to easily construct all necessary values for a PCS of stochastic policies by using a specific type of stochastic policy called a *mixture policy* (Vamplew et al., 2009). Furthermore, we will show that we can restrict our attention to deterministic policies when we have cooperative MODPs for constructing a CCS. In other words, the CCS

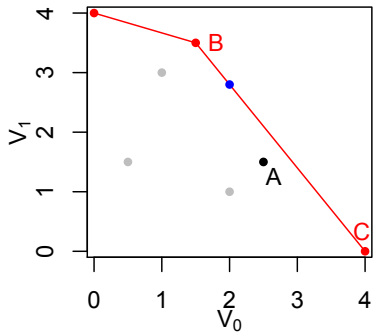


Figure 2.3: The CCS of deterministic stationary policies, mixture policies and the PCS of stochastic/non-stationary policies.

of deterministic policies, $CCS(\Pi^D)$, applies to cooperative MODPs when either, the scalarization function is linear, or policies can be stochastic, or both.

In scenarios in which multiple policies are required, e.g., the unknown weights and decision support scenarios of Section 1.1, where stochastic policies are allowed, we refer to the full set of all possible stochastic policies for an MODP as Π. However, when we can make the following assumption, we do not require all of Π to establish a CCS.

Assumption 1. Optimality of Deterministic Policies for Scalarized Instances *For any given* $\mathbf{w}$ *of a linear scalarization function (Definition 6), an MODP can be* scalarized *resulting in an SODP for which there is an optimal deterministic policy.*

If this assumption holds, we can employ stochastic policies instead of deterministic non-stationary ones. In particular, we can employ a *mixture policy* (Vamplew et al., 2009) π_m that takes a set of N deterministic policies, and selects the i-th policy from this set, π_i with probability p_i, where $\sum_{i=0}^{N} p_i = 1$. This leads to policy values that are a linear combination of the values of the constituent policies:

$$\mathbf{V}^{\pi_m} = \sum_{i=0}^{N} p_i \mathbf{V}^{\pi_i}.$$

When we consider the possible values we can attain through using these mixture policies on policies that are in the CCS (using the same values as for Figure 2.1) in Figure 2.3, we observe that we can create all the policy values on the red lines connecting the red dots (such as B and C, which represent the CCS policy values). This is a highly useful observation, as — as we discuss in the following chapters — in all the cooperative MODPs we consider, we can restrict ourselves to deterministic policies for computing a CCS.

Corollary 1. *(Vamplew et al., 2009; Roijers et al., 2013a) In an MODP for which Assumption 1 holds, there exists a* $CCS(\Pi^D)$ *that includes only deterministic policies, such that this set* P_M*, is a* $PCS(\Pi)$.

Proof. We can construct a policy with any value vector on the convex surface, e.g., the red lines connecting the red dots (which represent the CCS policy values) in Figure 2.3, by mixing policies on a CCS.[7] Therefore, we can always construct a mixture policy that dominates a policy, with a value under this surface, such as A. Furthermore, we show by contradiction that there cannot be any policy above the convex surface. If there was, it would be optimal for some $\mathbf{w}$ if f was linear. Consequently, due to Assumption 1, there would be a deterministic policy with at least equal value. But since the convex surface spans the values on the CCS, this leads to a contradiction. Therefore, no policy can Pareto-dominate a mixture policy on the convex surface. □

Thanks to Corollary 1, it is sufficient to compute a CCS, $CCS(\Pi^D)$, of deterministic policies to solve cooperative MODPs even when the scalarization function is non-linear, as long as we can establish that Assumption 1 holds. We show this for all MODPs of this dissertation in the following chapters. This leads us to the taxonomy of Table 2.1, in which for each scenario we discussed in Section 1.1 and type of policies allowed, for the different assumptions about the family of scalarization functions discussed in Section 2.1, the appropriate solution concept is provided. In this dissertation we focus on the case in which multiple policies are required (the unknown weights and decision support scenarios) and where either the scalarization function can be assumed to be linear, or the policies can be stochastic, or both. The corresponding solution concept is the CCS of deterministic policies, as highlighted in blue in the table.

In the rest of this dissertation, we focus on finding methods for computing the CCS of deterministic policies. For convenience, we typically assume linear f. However, please note that our methods also apply to merely monotonically increasing f, whenever stochastic policies are allowed.

[7] Note that we should always mix policies that are "adjacent" (such as B and C); the line between any pair of the policies we mix should be on the convex surface.

	single policy (known weights)		*multiple policies (unknown weights/decision support)*	
	deterministic	stochastic	deterministic	stochastic
linear scalarization	one deterministic policy		CCS of deterministic policies	
monotonically increasing scalarization	one deterministic policy	one mixture policy of two or more deterministic policies	PCS of deterministic policies	CCS of deterministic policies

Table 2.1: The MODP problem taxonomy — the columns describe whether the problem necessitates a single policy or multiple policies (and in which scenarios it does so), and whether those policies must be deterministic (by specification) or are allowed to be stochastic. The rows describe whether the scalarization function is a linear (Definition 6), or whether this cannot be assumed and the scalarization function is merely a monotonically increasing function. The contents of each cell describe what solution set should be used as a solution concept (as defined in Section 2.1).

Chapter 3

Optimistic Linear Support

In this chapter we present one of our central contributions: the *optimistic linear support (OLS)* framework for cooperative *multi-objective decision problems (MODPs)*. Before we go into OLS, we first discuss two approaches to solving MODPs in Section 3.1, which we refer to as the *inner* and the *outer loop* approaches. In the former, a single-objective algorithm for a specific decision problem (such as a CoGs, MDPs or POMDPs) is adapted to apply to the corresponding MODP, by changing the summation and maximization operators into cross-sum and suitable pruning operators. In the latter — to which OLS belongs — an MODP is solved as a series of scalarized (i.e., single-objective) problems, and single-objective algorithms are used as subroutines.

In Section 3.2, we analyze the outer loop approach, by identifying the similarities between computing a CCS for MODPs, and computing the value function of a (single-objective) POMDP. We observe that the *scalarized value function* of MODPs and the value function of single-objective POMDPs exhibit the same favorable property, which can be exploited in a similar way. However, we also observe that in the analogy between MODPs and POMDPs, the number of states in a POMDP corresponds to the number of objectives in an MODP. While the scalability in the number of states in a POMDP is typically the bottleneck, the number of objectives in an MODP is typically much smaller and therefore not the bottleneck. In fact, there are many real-world problems with two or three objectives. Therefore, an algorithm that is efficient for POMDPs with only a small number of states but does not scale well, may still be a good starting point for creating an MODP algorithm.

In Section 3.3, we define the OLS framework, and explain the algorithm in detail. OLS is a generic outer loop method, that takes inspiration from the POMDP literature. It takes a single-objective method as a subroutine, and calls this method a finite number of times in order to solve an MODP optimally. In Section 3.4, we analyze OLS theoretically, and show that OLS has many advantages over inner loop algorithms, that extend the same single-objective algorithms used by OLS as subroutines. Firstly, OLS comes with strong guarantees with respect to time and space complexity. Secondly, OLS-based algorithms can be much faster for small and medium numbers of objectives than

A_l	$(5.7, 6.9)$
B_l	$(7.1, 5.7)$
C_l	$(7.5, 5.4)$
D_l	$(6.6, 6.7)$

$\mathbf{V}_l(a_l)$

A_r	$(7.3, 7.6)$
B_r	$(5.9, 8.2)$
C_r	$(8.8, 6.4)$
D_r	$(6.6, 7.7)$

$\mathbf{V}_r(a_r)$

Table 3.1: A simple MODP — select one element from each list and receive the associated reward vector.

corresponding inner loop algorithms. And finally, in OLS, the single-objective subroutines can be used out of the box[1], making any improvement for single-objective methods an improvement for multi-objective methods.

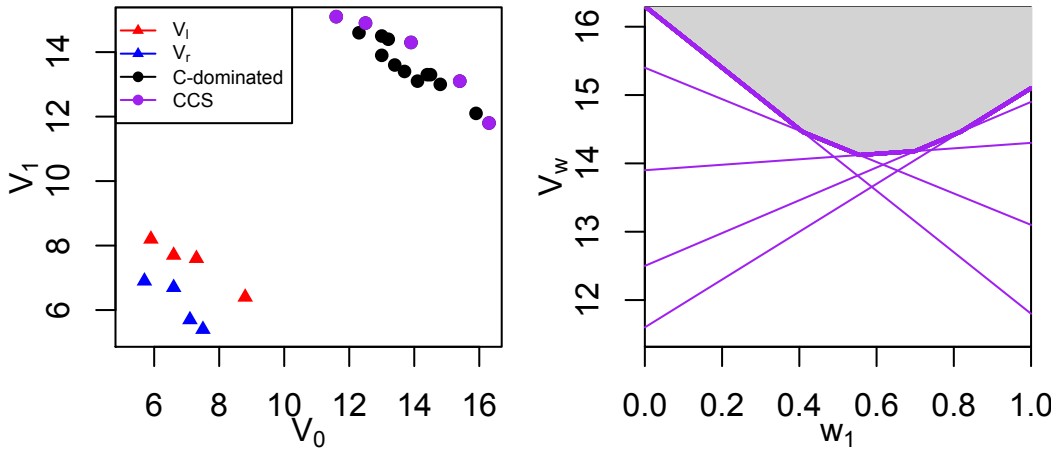
After defining and analyzing OLS, we make two key improvements to OLS, that improve its applicability for different MODPs. In Section 3.5, we show that OLS can be used in combination with bounded approximate single-objective subroutines, and show that if this is the case, OLS produces ε-CCSs (Definition 13). In Section 3.6, we show that we can improve the runtime of OLS, by reusing the solutions found by earlier calls to the single-objective subroutines to hot-start later calls to these subroutines.

3.1 Inner Loop versus Outer Loop

In this section, we present a "how-to" on creating algorithms that compute a CCS for an MODP, starting from a method that computes the optimal policy (and associated value) for the corresponding single-objective decision problem (SODP). We discuss two popular approaches. The first, which we refer to as the *inner loop approach*, adapts the inner workings of the single-objective algorithm by exchanging sums and maximizations by cross-sums and pruning, which we will define soon. The second, which we refer to as the *outer loop approach*, leaves the single-objective algorithm intact, but creates a shell, i.e., an outer loop, around the single-objective method.

In order to illustrate the difference between inner and outer loop methods, we make use of the following simple multi-objective decision problem (Table 3.1): there are two local payoff functions called $\mathbf{V}_l(a_l)$ and $\mathbf{V}_r(a_r)$ with for each local action (A, B, C, or D), an expected reward. A deterministic policy takes one action a_l and one action a_r. The value of a deterministic policy is the sum of the local rewards. We denote a deterministic policy as $\mathbf{a} = (a_l, a_r)$ (a joint action), and the value of a deterministic policy as $\mathbf{V}(\mathbf{a}) = \mathbf{V}_l(a_l) + \mathbf{V}_r(a_r)$. For example, for the deterministic policy $\mathbf{a} = (A_l, A_r)$, the value would be $\mathbf{V}(A_l, A_r) = (13, 14.5)$.

[1] Though sometimes, adaptations can be made as we explain in Section 5.3.



(a) (b)

Figure 3.1: (a) All values and local rewards for the problem of Table 3.1. (b) The optimal scalarized value function, $V^*_{CCS}(\mathbf{w}) = \max_{\mathbf{V} \in CCS} \mathbf{w} \cdot \mathbf{V}$.

A single-objective version of our toy problem would consist of two local payoff functions with scalar values. When we are given a specific linear scalarization weight $\mathbf{w}$, we can scalarize the problem of Table 3.1, resulting in such a single-objective problem. For example, when $\mathbf{w} = (0.5, 0.5)$, the rewards in $\mathbf{V}_l$ would become: $V_{l,\mathbf{w}}(A_l) = 6.3, V_{l,\mathbf{w}}(B_l) = 6.4, V_{l,\mathbf{w}}(C_l) = 6.45$, and $V_{l,\mathbf{w}}(D_l) = 6.65$. From such a local payoff function, it is easy to pick the maximizing local action. Furthermore, combining the maximizing actions from both lists provides the maximizing policy for the full problem. A single-objective solver would thus be:

1. pick the maximum from each list, and
2. compute the sum.

Finding the maximum value can thus be expressed as:

$$\max_{\mathbf{a}} V_{\mathbf{w}}(\mathbf{a}) = \max_{a_l} V_{l,\mathbf{w}}(a_l) + \max_{a_r} V_{r,\mathbf{w}}(a_r), \tag{3.1}$$

and the maximizing policy can be found by using $\arg\max$ instead of $\max$. When we call the number of elements in each list l, the single-objective algorithm returns the optimal solution in $O(l)$ time.

For the multi-objective version (Table 3.1), we want to compute the CCS. A naive way to do so would be to first list all possible joint actions and associated value vectors and compute the CCS from that list. The multi-objective values of all possible deterministic policies are illustrated in black and purple in Figure 3.1. Because there are $l^2 = 16$ deterministic policies, but only 5 CCS policies, we might be doing a lot of unnecessary work when we follow this approach. In the following two subsections, we

provide two approaches to adapting single-objective methods in a smart way. In the problem of Table 3.1, *both* approaches avoid full enumeration of deterministic policies.

3.1.1 The Inner Loop Approach

A key difference between SODPs and MODPs, is that while in single-objective problems the solution is a single policy that maximizes a scalar value, in multi-objective problems, the solution is typically a set of policies and associated value vectors, i.e., coverage sets. For convenience we assume that the CSs contain both value vectors as well as the policies that achieve these value vectors. We do not go into the implementation details of how this is achieved in this section.[2]

In order to work with solution sets, we first need to translate the problem to sets rather than local reward functions:

$$\mathcal{V}_l = \{\mathbf{V}_l(a_l) : a_l \in \{A_l, B_l, C_l, D_l\}\},$$

and accordingly for $\mathbf{V}_r$.

In the *inner loop approach*, the required solution sets are produced by changing the operators required for optimization in single-objective problems, to operators that work on sets. The first single-objective operator that we must adapt in the inner loop approach is the maximization operator. In the multi-objective setting the maximization corresponds to computing a (local) coverage set. Computing a CS can be seen as computing the maximal value *for all weights* of the scalarization function *in parallel*. The multi-objective operator corresponding to maximization takes a set, such as $\mathcal{V}_l$, and *prunes* away all vectors that do not maximize the value for any weight $\mathbf{w}$ of the scalarization function f, resulting in a local CS. Therefore, we call these operators *pruning* operators. Note that when computing a PCS, we need a different pruning operator than when computing a CCS. In its abstract form, i.e., when the operator computes a local CS (either a CCS or PCS), we denote this operator as LCS.

The second single-objective operator we must adapt is the summation operator. The corresponding multi-objective operator is the *cross-sum* operator, $\oplus$, which combines two sets by computing the sum of all possible pairs with one element from each set:

$$\mathcal{A} \oplus \mathcal{B} = \{\mathbf{A} + \mathbf{B} : \mathbf{A} \in \mathcal{A} \wedge \mathbf{B} \in \mathcal{B}\}. \tag{3.2}$$

Having defined the appropriate operators, we can now translate Equation 3.1, to reflect the inner loop method for the problem of Table 3.1:

$$CS = LCS(LCS(\mathcal{V}_l) \oplus LCS(\mathcal{V}_r)). \tag{3.3}$$

Note that while there are only two maximization operations in Equation 3.1, there are now three LCS operations. This is because while the sum of maximal local rewards

[2] We do go into these details for the specific MODPs we discuss in later chapters.

in the single-objective problem form a single value when summed, the combination of two sets may contain combinations that are dominated.

We have now defined the general schema for creating a multi-objective inner loop method from a base single-objective method, i.e., replace the maximizations by pruning operations, and replace the summations by cross-sums. Before we can use such a method in practice however, we must:

(a) define appropriate implementations of LCS depending on what type of solution set we are after, e.g., a CCS or a PCS,

(b) select the places in the algorithm where pruning is applied, and

(c) check whether the resulting inner loop method is indeed correctly outputting the CS. In particular, we check that no *necessary* vectors for computing the CS pruned prematurely, and that no excess vectors, i.e., vectors that are never optimal for any allowed scalarization, retained in the output.

First, let us address (a). We provide a possible implementation of LCS for the case of CCS computations in Algorithm 1. We use this implementation throughout this dissertation, and refer to it as `CPrune`. `CPrune` is based on the algorithm for CCS pruning[3] by Feng and Zilberstein (2004), with one modification. In order to improve runtime guarantees, `CPrune` first pre-prunes the input set, $\mathcal{V}$, to a PCS using the `PPrune` algorithm (Algorithm 2) at line 1. `PPrune` computes a PCS in $O(d|\mathcal{V}||PCS|)$ time by running pairwise comparisons. Next, a partial CCS, $\mathcal{V}^*$, is constructed as follows: a random vector $\mathbf{V}$ from $\mathcal{V}'$ is selected at line 4. For $\mathbf{V}$ the algorithm tries to find a weight vector $\mathbf{w}$ for which $\mathbf{V}$ is better than the vectors in $\mathcal{V}^*$ (line 5), by solving the linear program in Algorithm 3. If there is such a $\mathbf{w}$, `CPrune` finds the best vector $\mathbf{V}'$ for $\mathbf{w}$ in $\mathcal{V}'$ and moves it to $\mathcal{V}^*$ (line 10–12). If there is no weight for which $\mathbf{V}$ is better, it is C-dominated and thus removed from $\mathcal{V}'$ (line 7).

Theorem 2. *The computational complexity of* `CPrune` *as defined by Algorithm 1 is*

$$O(d|\mathcal{V}||PCS| + |PCS|P(d|CCS|)), \tag{3.4}$$

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).

Concerning (c), i.e., whether the algorithm outputs the correct CS, we provide a small proof sketch for our simple MODP problem of Table 3.1. (In later chapters we provide more formal proofs.) First, note that for each $\mathbf{w}$, `CPrune` retains at least one optimal value vector, and removes all vectors that are not optimal for any $\mathbf{w}$. Because

[3]The work by Feng and Zilberstein (2004) is from the POMDP literature. We explain the relation between POMDPs and multi-objective decision making in Section 3.2. Note that many algorithms for CCS pruning originate in other fields than POMDPs, such as graphics or geometry (e.g., Graham (1972)).

Algorithm 1: `CPrune`$(\mathcal{V})$

Input: A set of value vectors $\mathcal{V}$

$\mathcal{V}' \leftarrow$ `PPrune`$(\mathcal{V})$
$\mathcal{V}^* \leftarrow \emptyset$
while $\mathcal{V}' \neq \emptyset$ **do**
  select random $\mathbf{V}$ from $\mathcal{V}'$
  $\mathbf{w} \leftarrow$ `findWeight`$(\mathbf{V}, \mathcal{V}^*)$; `// identify a` $\mathbf{w}$ `where` $\mathbf{V}$ `improves` $\mathcal{V}^*$
  if $\mathbf{w}$=*null* **then**
    remove $\mathbf{V}$ from $\mathcal{V}'$; `// there is no` $\mathbf{w}$ `where where` $\mathbf{V}$ `improves` $\mathcal{V}^*$
  end
  else
    $\mathbf{V}' \leftarrow \arg\max_{\mathbf{V} \in \mathcal{V}'} \mathbf{w} \cdot \mathbf{V}$; `// find best value vector for` $\mathbf{w}$
    $\mathcal{V}' \leftarrow \mathcal{V}' \setminus \{\mathbf{V}'\}$
    $\mathcal{V}^* \leftarrow \mathcal{V}^* \cup \{\mathbf{V}'\}$
  end
end
return $\mathcal{V}^*$

Algorithm 2: `PPrune`$(\mathcal{V})$

Input: A set of value vectors $\mathcal{V}$

$\mathcal{V}^* \leftarrow \emptyset$
while $\mathcal{V} \neq \emptyset$ **do**
  $\mathbf{V} \leftarrow$ the first element of $\mathcal{V}$
  foreach $\mathbf{V}' \in \mathcal{V}$ **do**
    if $\mathbf{V}' \succ_P \mathbf{V}$ **then**
      $\mathbf{V} \leftarrow \mathbf{V}'$; `// Continue with` $\mathbf{V}'$ `instead of` $\mathbf{V}$
    end
  end
  Remove $\mathbf{V}$, and all vectors P-dominated by $\mathbf{V}$, from $\mathcal{V}$
  Add $\mathbf{V}$ to $\mathcal{V}^*$
end
return $\mathcal{V}^*$

Algorithm 3: `findWeight`$(\mathbf{V}, \mathcal{V})$

`// Find a weight vector` $\mathbf{w}$ `, where the scalarized value of a new value vector` $\mathbf{V}$ `is an improvement, i.e.,` $\mathbf{w} \cdot \mathbf{V} > \max_{\mathbf{V}' \in \mathcal{V}} \mathbf{w} \cdot \mathbf{V}'$ `, using a linear program.`

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

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

linear scalarization distributes over addition, i.e., $\mathbf{w} \cdot (\mathbf{V}_l(a_l) + \mathbf{V}_r(a_r)) = \mathbf{w} \cdot \mathbf{V}_l(a_l) + \mathbf{w} \cdot \mathbf{V}_r(a_r)$, we know that no optimal values can be lost when performing `CPrune` on the separate sets $\mathcal{V}_l$ and $\mathcal{V}_r$, before taking the cross-sum. Furthermore, we know that because `CPrune` is applied again after taking the cross-sum, no excess value vectors can remain.

When we apply the inner loop method, i.e., Equation 3.3 where $LCS =$ `CPrune`, to the problem of Table 3.1 (and Figure 3.1(a)), we observe the following: when applying `CPrune` to $\mathcal{V}_l$ and $\mathcal{V}_r$, it removes one vector each. In other words, the local CCSs both consist of three vectors. Therefore, the cross-sum of these CCSs consists of 9 vectors, which is significantly less than the 16 we would get if we took the cross-sum before local pruning. However, in order to do the local pruning, we have to invest effort, which in this (small) problem amounts to as much work as computing the cross-sum. Therefore, we have to be careful when applying the inner loop approach; it is not always worth the effort to prune everywhere we can. Selecting how much to prune — by choosing either to not prune at all, apply only `PPrune`, or to apply `CPrune`, at each possible point in the algorithm — is a non-trivial design choice that may be very problem-specific. In Sections 4.3.1 and 4.3.3 for example, we analyze the different design choices for an inner loop method for MO-CoGs.

Many algorithms in the literature on multi-objective decision problems, take an inner loop approach (e.g., the methods by Rollón (2008) and Wilson et al. (2015) for MO-CoGs, and Barrett and Narayanan (2008) and Moffaert and Nowé (2014) for MOMDPs). However, it is also possible to create multi-objective methods by creating a shell around a single-objective method, rather than making major changes to the single-objective method itself, as we describe in the next subsection.

3.1.2 The Outer Loop Approach

In an *outer loop approach*, an (approximate) coverage set is built incrementally, by solving scalarized instances. In order to solve these scalarized instances, a subroutine appropriate for the single-objective version of the MODP at hand is required. For example, when we want to solve an MOMDP, an MDP solver is required as a subroutine. The solutions produced for scalarized instances are kept in a *partial CCS*.

Definition 16. *A* partial CCS*, $\mathcal{S}$, is a subset of a CCS, which is in turn a subset of all possible value vectors ($\mathcal{V} = \{\mathbf{V}^\pi : \pi \in \Pi\}$), i.e., $\mathcal{S} \subseteq CCS \subseteq \mathcal{V}$.*

The basic structure of an outer loop method is given in Algorithm 4. It starts from an empty set (line 1), as a partial CCS. In each iteration, a scalarized instance (i.e., an SODP resulting from scalarizing the MODP with a given f and a $\mathbf{w}$) is selected (line 3), and solved using a single-objective subroutine (line 4). For each solution, i.e., an optimal policy $\pi^*_\mathbf{w}$, of a scalarized instance, the multi-objective value vector $\mathbf{V}_\mathbf{w}$ is retrieved. If $\mathbf{V}_\mathbf{w}$ improves upon the partial CS, $\mathcal{S}$ — by improving the scalarized value for *some* (allowed) f and $\mathbf{w}$ — it is added to $\mathcal{S}$ (lines 5–8). This process continues until some stop criterion is reached, e.g., when it can be proven that $\mathcal{S}$ is a CS or time runs out.

Algorithm 4: `OuterLoopMethod`$(m,$ `SolveSO`$)$

Input: An MODP, m, and a corresponding single-objective solver `SolveSO`.

1 $\mathcal{S} \leftarrow \emptyset$; `// a partial CS`
2 **while** *stop criterion not reached* **do**
3 $m_\mathbf{w} \leftarrow$ select a $\mathbf{w}$ and scalarize m
4 $\pi^*_\mathbf{w} \leftarrow$ `SolveSO`$(m_\mathbf{w})$
5 $\mathbf{V}_\mathbf{w} \leftarrow$ retrieve/compute the multi-objective value of $\pi^*_\mathbf{w}$
6 **if** $\mathbf{V}_\mathbf{w}$ *improves upon* S **then**
7 $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{V}_\mathbf{w}\}$; `// add` $\mathbf{V}_\mathbf{w}$ `(and associated` $\pi^*_\mathbf{w}$`) to` $\mathcal{S}$`.`
8 **end**
9 **end**
10 **return** S

The three main design choices when creating an outer loop method are:

(a) how to select scalarized instances to solve,

(b) which single-objective subroutine to use, and

(c) whether to compute the value vectors separately (by policy evaluation), or to adapt the single-objective solver such that it returns both $\pi^*_\mathbf{w}$ and $\mathbf{V}_\mathbf{w}$.

For example, in *random sampling (RS)* (which we use as a baseline in Roijers et al. (2015c)), scalarized instances are selected randomly, by sampling one allowed $\mathbf{w}$ (and f) at each iteration and scalarizing the problem. The scalarized problem is then solved

using, e.g., an out-of-the-box single-objective solver that produces an optimal policy for $m_{\mathbf{w}}$, and the value vector, $\mathbf{V}_{\mathbf{w}}$ retrieved by a standard policy evaluation algorithm. At each iteration, it is checked whether $\mathbf{V}_{\mathbf{w}}$ is already in $\mathcal{S}$ or not — note that $\mathbf{V}_{\mathbf{w}}$ cannot be dominated when the single-objective solver is optimal, because it is optimal for at least one scalarized instance by definition — and if not, added to $\mathcal{S}$. Typically, this is done for a limited number of iterations, or until a fixed number I_{stop} of iterations has not produced a new value vector for $\mathcal{S}$.

RS has two important advantages. Firstly, it quickly produces *some* result, and improves upon it iteratively, i.e., it is an *anytime* algorithm. This is a major advantage over many inner loop methods, as they typically need to run until completion to produce any useful results. Secondly, RS is extremely easy to implement, as long as a suitable single-objective solver and policy evaluation method are available. However, RS also has a major downside: it is optimal only in the limit. Because of its randomized nature, there is no guarantee that (within finite time) there will be no f and $\mathbf{w}$ left for which an improvement still exists.

More sophisticated linear scalarization weight sampling-based methods (Kim and de Weck, 2005; Van Moffaert et al., 2014) exist (in those articles referred to as *weighted-sum* methods), in which scalarizations are sampled on the basis of, e.g., the relative distance of the value vectors $\mathbf{V}$ or the hyper-volume metric. However, these methods also do not provide runtime and optimality guarantees.

In Section 3.3, we present a novel algorithm which retains the advantages of RS and other outer loop methods, but is provably optimal when the single-objective subroutines used are optimal, and produces bounded approximations when the subroutines produce upper and lower bounds, and runs in finite time. In order to do this, we make use of an analogy between finding the CCS for MODPs, and POMDP planning.

3.2 The Scalarized Value Function

In order to create outer loop methods that are provably optimal, run in finite time, and make smart choices about which scalarized instances of the MODP to solve next, we first need to define the *scalarized value function* with respect to the linear scalarization function (Definition 6). Note that focussing on linear scalarization is sufficient to discover the CCS, even though the CCS can also be used in the context of non-linear scalarization.

When we look at the linearly scalarized value $\mathbf{V}_{\mathbf{w}}$ of a value vector $\mathbf{V}$ as a function of $\mathbf{w}$, it becomes a hyperplane above the weight simplex. For example, the scalarized values for the vectors in the CCS of Table 3.1 are shown in Figure 3.1b. Because this is a 2-objective problem, the weight for objective 0, w_0, is equal to $1 - w_1$. When we plot the scalarized value as a function of w_1, each value vector becomes a line.

The maximal scalarized value function over a set of vectors, $\mathcal{S}$, takes the maximum over all the vectors in that set.

Definition 17. *A scalarized value function over a partial CCS, $\mathcal{S}$, is a function that*

takes a weight vector $\mathbf{w}$ *as input, and returns the maximal attainable scalarized value with any payoff vector in* S*:*

$$V^*_{\mathcal{S}}(\mathbf{w}) = \max_{\mathbf{V}^\pi \in \mathcal{S}} \mathbf{w} \cdot \mathbf{V}^\pi.$$

If we take a complete CCS as $\mathcal{S}$, the scalarized value function $V^*_{CCS}(\mathbf{w})$ is the *optimal* scalarized value function. By definition, for every $\mathbf{w}$ the optimal scalarized value for the MODP is $V^*_{CCS}(\mathbf{w})$. In Figure 3.1(b), the optimal scalarized value function is represented by the bold line segments. Similar to the scalarized value function, we define the set of maximizing value vectors and associated policies:

Definition 18. *The optimal value vector set function with respect to* $\mathcal{S}$ *is a function that gives the value vectors that maximize the scalarized value for a given* $\mathbf{w}$*:*

$$\mathcal{V}_{\mathcal{S}}(\mathbf{w}) = \arg\max_{\mathbf{V}^\pi \in \mathcal{S}} \mathbf{w} \cdot \mathbf{V}^\pi,$$

Similarly, the optimal policy set $\Pi_{\mathcal{S}}(\mathbf{w})$*, is the set of policies that constitute the value vectors in* $\mathcal{V}_{\mathcal{S}}(\mathbf{w})$*.*

Note that $\mathcal{V}_S(\mathbf{w})$ and $\Pi_{\mathcal{S}}(\mathbf{w})$ are sets because for some $\mathbf{w}$ there can be multiple value vectors that provide the same scalarized value.

Because $V^*_{\mathcal{S}}(\mathbf{w})$ is the maximum scalarized value for each $\mathbf{w}$, it is the convex upper surface of all of these lines (which represent the value vectors in $\mathcal{S}$). Hence, $V^*_{\mathcal{S}}(\mathbf{w})$ and $V^*_{CCS}(\mathbf{w})$ are *piecewise linear and convex* (PWLC) functions.

When considering *partially observable Markov decision processes* (POMDPs) (Cheng, 1988; Kaelbling et al., 1998), the optimal value function is also a PWLC function. In particular, the belief vectors b in POMDPs correspond to our weight vectors $\mathbf{w}$ and the α-vectors correspond to our value vectors $\mathbf{V}$. POMDPs and MODPs are thus related problems. This is an important observation, as many insights from the POMDP literature can thus also be exploited in the context of computing CCSs for MODPs. However, while scalability in the number of states in a POMDP is key to the succes of a POMDP solver, the number of objectives in an MODP is typically small — there are many MODPs with only two or three objectives. Therefore, scalability in the number of objectives is typically less important than scalability in other properties of an MODP. In the next section, we build off a POMDP method that has poor scalability in the number of POMDP states, but forms a good starting point for creating an MODP method.

3.3 The OLS Algorithm

In this section, we present our main algorithmic contribution: *optimistic linear support* (OLS). OLS is a novel outer loop algorithm for computing the CCS for MODPs.

Like the random sampling (RS) method described in Section 3.1.2, OLS deals with multiple objectives in the *outer loop* by employing a single-objective method as a subroutine, and building the CCS incrementally. With each iteration of its outer loop, OLS thus adds at most one new vector to a partial CCS, $\mathcal{S}$. To find this vector, OLS selects a single linear scalarization weight, $\mathbf{w}$. The key difference with RS is that OLS selects this $\mathbf{w}$ intelligently. Specifically, OLS is optimistic: it selects the $\mathbf{w}$ that offers the maximal possible improvement — an upper bound on the difference between $V^*_{\mathcal{S}}(\mathbf{w})$ and the optimal scalarized value function $V^*_{CCS}(\mathbf{w})$. After OLS identifies it, this $\mathbf{w}$ is passed to the inner loop. In the inner loop, OLS calls a problem specific single-objective solver to solve the single-objective decision problem that results from scalarizing the MODP using the $\mathbf{w}$ selected by the outer loop. The policy that is optimal for this scalarized problem and its value vector are then added to the partial CCS.

The departure point for creating OLS is *Cheng's linear support (CLS)* (Cheng, 1988). CLS 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 an MODP, and while realistic POMDPs typically have many states, many MODPs have only a handful of objectives. Therefore, for MODPs, scalability in the number of objectives is typically less important than scalability in other properties of the problem (such as the number of agents or the number of states), making Cheng's linear support an attractive starting point for developing efficient MODP algorithms.

Because OLS takes an arbitrary single-objective problem solver as input, it can be seen as a generic multi-objective method that applies to any cooperative MODP. 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. After defining the algorithm in this section, we show that OLS inherits any favorable properties from the single-objective subroutines it uses in Section 3.4. In subsequent chapters, we instantiate OLS by using different single-objective solvers for different instances of MODPs, and show how this leads to novel state-of-the-art algorithms for a variety of problems.

In this section, we assume that the single-objective solver used as a subroutine by OLS is exact, i.e., the scalarized instances are solved optimally. In other words, we assume that OLS has access to a function called `SolveSODP` that computes the best payoff vector for a given $\mathbf{w}$. For now, we leave the implementation of `SolveSODP` abstract. As mentioned in Section 3.1.2, we can either use an adapted single-objective solver that returns the value vector, or we can use a separate policy evaluation step. In Section 3.5 we relax the assumption that `SolveSODP` needs to be exact.

OLS exploits the PWLC property of the scalarized value function, $V^*_{\mathcal{S}}(\mathbf{w})$ (Definition 17). OLS incrementally builds up the CCS, by adding new value vectors that are found at so-called *corner weights*, to an initially empty partial CCS, $\mathcal{S}$. These corner weights are the weights where $V^*_S(\mathbf{w})$ changes slope in all directions. These must thus

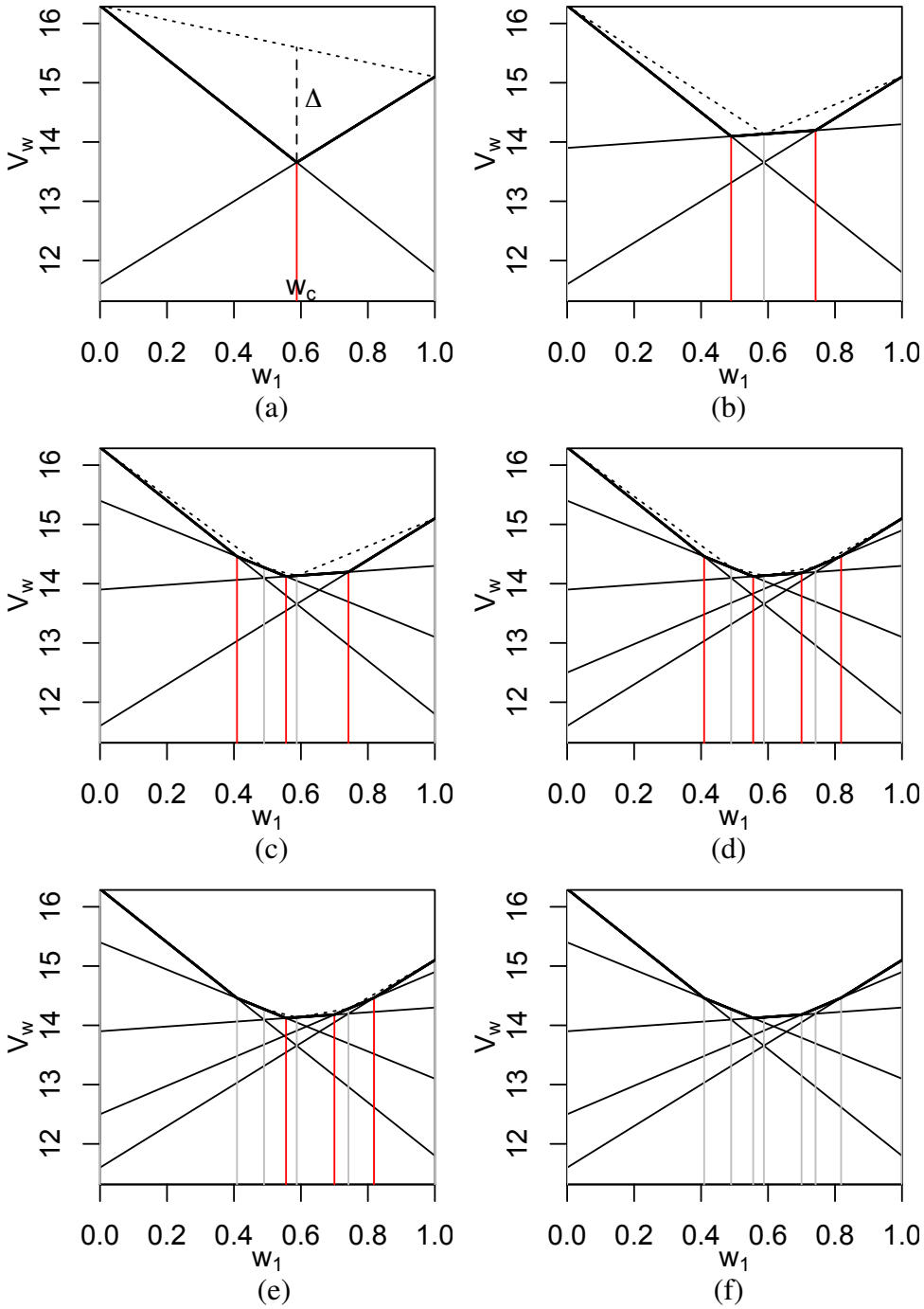


Figure 3.2: A run of OLS for the problem of Table 3.1. In each picture, V_S^* is indicated by bold line segments, corner weights are indicated by vertical lines (gray for visited, and red for pending), and $\overline{CCS}$ is shown as dotted lines. (a) OLS finds two payoff vectors at the extrema, and a new corner weight $\mathbf{w}_c = (0.4125, 0.5875)$ is found, with maximal possible improvement Δ. (b) OLS finds a new vector at $\mathbf{w}_c$, and adds two new corner weights to Q. (c and d) OLS finds another new vector and adds two new corner weights to Q. (e) OLS does not find a new vector for the corner weight it selects, reducing the maximal possible improvement at that corner weight to 0. (f) For the remaining corner weights, no new vectors are found, ensuring $S = \overline{CCS} = CCS$.

Algorithm 5: `OLS`$(m, \texttt{SolveSODP}, \varepsilon)$

Input: A MODP m, a single-objective subroutine `SolveSODP`, and max. allowed error ε

$\mathcal{S} \leftarrow \emptyset$; `// a partial CCS`
$\mathcal{W} \leftarrow \emptyset$; `// a set of visited weights`
$Q \leftarrow$ an empty priority queue
foreach *extremum of the weight simplex* $\mathbf{w}_e$ **do**
  Q.add($\mathbf{w}_e$, ∞) ; `// add the extrema to Q with infinite priority`
end
while $\neg$*Q.isEmpty()* $\wedge$ $\neg$*timeOut* **do**
  $\mathbf{w} \leftarrow$ Q.pop()
  $\mathbf{V}^\pi \leftarrow \texttt{SolveSODP}(m, \mathbf{w})$
  $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathbf{w}\}$
  if $\mathbf{V}^\pi \notin \mathcal{S}$ **then**
    $W_{del} \leftarrow$ remove the corner weights made obsolete by $\mathbf{V}$ from Q, and store them
    $W_{del} \leftarrow \{\mathbf{w}\} \cup W_{del}$; `// corner weights to remove`
    $W_{\mathbf{V}^\pi} \leftarrow \texttt{newCornerWeights}(\mathbf{V}^\pi, W_{del}, \mathcal{S})$
    $\mathcal{S} \leftarrow S \cup \{\mathbf{V}^\pi\}$
    foreach $\mathbf{w} \in W_{\mathbf{V}^\pi}$ **do**
      $\Delta_r(\mathbf{w}) \leftarrow$ calculate improvement using $\texttt{maxValueLP}(\mathbf{w}, \mathcal{S}, \mathcal{W})$
      if $\Delta_r(\mathbf{w}) > \varepsilon$ **then**
        Q.add($\mathbf{w}$, $\Delta_r(\mathbf{w})$)
      end
    end
  end
end
return $\mathcal{S}$ *and the highest* $\Delta_r(\mathbf{w})$ *left in* Q

be weights where $\mathcal{V}_{\mathcal{S}}(\mathbf{w})$ and $\Pi_{\mathcal{S}}(\mathbf{w})$ (Definition 18) consist of multiple value vectors and associated policies. Every corner weight is prioritized by the maximal possible improvement of finding a new value vector at that corner weight. When the maximal possible improvement is 0, OLS knows that the partial CCS is complete.

As an example of this process, we show the entire run of OLS for the problem of Table 3.1 in Figure 3.2. The (corner) weights where the algorithm has already searched for new payoff vectors are indicated by gray vertical lines, and the corner weights that are still pending are indicated by red vertical lines.

OLS is shown in Algorithm 5. OLS takes as input: m, the MODP to be solved, the single-objective subroutine, `SolveSODP`, and ε, the maximal tolerable error in the result.

We first describe how OLS is initialized. Then, we define corner weights formally and describe how OLS identifies them. Finally, we describe how 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.

Initialization

OLS starts by initializing the partial CCS, $\mathcal{S}$, which contains the payoff vectors in the CCS that have been discovered so far (line 1 of Algorithm 5), as well as the set of visited weights $\mathcal{W}$ (line 2). Then, OLS 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).

The First Iterations

The extrema that we added in the initialization 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). Because the extrema have infinite priority, they are guaranteed to be popped off the queue first. `SolveSODP` is called with $\mathbf{w}$ (line 9) to find $\mathbf{V}$, the best payoff vector for that $\mathbf{w}$.

In the algorithm, we assume that `SolveSODP` returns the best value vector for $\mathbf{w}$. This can be done by adapting an out-of-the-box single-objective solver to return this value vector, or by adding a separate policy evaluation step. In the latter case line 9 of the algorithm could instead be written as:

$\pi \leftarrow$ `SolveSODP`$(m, \mathbf{w})$
$\mathbf{V}^{\pi} \leftarrow$ `PolicyEval`(m, π)

Throughout this dissertation, we assume that we can determine the exact value of $\mathbf{V}^{\pi}$ of the π that `SolveSODP` returns, i.e., the policy evaluation step is exact.

To illustrate the situation after initialization, Figure 3.2a shows $\mathcal{S}$ after two payoff vectors of the 2-objective list combination problem of Table 3.1 have been found by applying `SolveSODP` to the extrema of the weight simplex. This leads to $S = \{(16.3, 11.8), (11.6, 15.1)\}$. Each of these vectors must be part of the CCS because it is optimal for at least one $\mathbf{w}$: the one for which `SolveSODP` returned it as a solution. The set of weights $\mathcal{W}$ that OLS has tested so far, i.e., the extrema of the weight simplex, are marked with vertical gray line segments.

Corner Weights

After having evaluated the extrema, $\mathcal{S}$ 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 $\mathbf{V}^{\pi}$ to add to S, OLS must determine which new weight vectors to add to Q, and with what priority. Like Cheng's linear support, OLS identifies the *corner weights*: the weights at the corners of the convex upper surface, i.e., the points where the PWLC surface $V^*_S(\mathbf{w})$ changes slope. To define the corner weights precisely, we define P, the polyhedral subspace that is above $V^*_{\mathcal{S}}(\mathbf{w})$ (Bertsimas and Tsitsiklis, 1997). For example, in Figure 3.1b, P is displayed as the shaded area above the optimal scalarized value function. The corner weights are the vertices of P, which can be defined by a set of linear inequalities:

Algorithm 6: `newCornerWeights`$(\mathbf{V}^{\pi}_{new}, W_{del}, \mathcal{S})$

Input: A new value vector, $\mathbf{V}^{\pi}_{new}$, a set of obsolete corner weights, W_{del}, and the current partial CCS, $\mathcal{S}$

1 $\mathcal{V}_{rel} \leftarrow \bigcup_{\mathbf{w} \in W_{del}} \mathcal{V}_{\mathcal{S}}(\mathbf{w})$; `// involved value vectors (see Definition 18)`
2 $\mathcal{B}_{rel} \leftarrow$ the set of boundaries of the weight simplex involved in any $\mathbf{w} \in W_{del}$;
3 $W_{new} \leftarrow \emptyset$; `// new corner weights`
4 **foreach** *subset* X *of* $d-1$ *elements from* $\mathcal{V}_{rel} \cup \mathcal{B}_{rel}$ **do**
5 $\mathbf{w}_c \leftarrow$ compute weight where the vectors/boundaries in X intersect with $\mathbf{V}^{\pi}_{new}$;
6 **if** $\mathbf{w}_c$ *is inside the weight simplex* **then**
7 **if** $\mathbf{w}_c \cdot \mathbf{V}^{\pi}_{new} = V^{*}_{\mathcal{S}}(\mathbf{w}_c)$ **then**
8 $W_{new} \leftarrow W_{new} \cup \mathbf{w}_c$
9 **end**
10 **end**
11 **end**
12 **return** W_{new}

Definition 19. *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 $\mathbf{x} = (w_1, ..., w_d, V_{\mathbf{w}})$ 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 $(w_1, ..., w_d, V_{\mathbf{w}})$.*

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 the new value vector $\mathbf{V}^{\pi}$, OLS identifies which corner weights change in the polyhedron P by adding $\mathbf{V}^{\pi}$ 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 $\mathbf{V}^{\pi}$ yields a better scalarized value than currently known are deleted from the queue (line 12) and then the function `newCornerWeights`$(\mathbf{V}^{\pi}, W_{del}, \mathcal{S})$ (line 14) calculates the new corner weights that involve $\mathbf{V}^{\pi}$.

The function `newCornerWeights`$(\mathbf{V}^{\pi}, W_{del}, \mathcal{S})$ (Algorithm 6) first calculates the set of all relevant payoff vectors, $\mathcal{V}_{rel}$, by taking the union of all the maximizing vectors of the weights in W_{del} for $\mathcal{S}$ (on line 1). In our implementation, we optimize this step by caching $\mathcal{V}_{\mathcal{S}}(\mathbf{w})$ (Definition 18) and associated policies for all $\mathbf{w}$ in Q. If for a corner weight, $\mathbf{w}$, $\mathcal{V}_{\mathcal{S}}(\mathbf{w})$ contains fewer than d value vectors, then a boundary of the weight simplex is involved. These boundaries are also stored (line 2). All possible subsets of size $d-1$ — of both vectors and boundaries — are taken. For each subset the weight where these $d-1$ payoff vectors (and boundaries) intersect with

each other and $\mathbf{V}$ is computed by solving a system of linear equations (on line 5). The intersection weights for all subsets together form the set of candidate corner weights: W_{can}. `newCornerWeights`$(\mathbf{V}^\pi, W_{del}, \mathcal{S})$ returns the subset of W_{can} which are inside the weight simplex and for which there is no vector $\mathbf{V}' \in \mathcal{S}$ that has a higher scalarized value than $\mathbf{V}$ at that weight. OLS then adds the new corner weights returned by `newCornerWeights`$(\mathbf{V}^\pi, W_{del}, \mathcal{S})$ to its queue, Q.

In Figure 3.2a, after computing the value vectors for the two extrema, one new corner weight is produced, labelled $\mathbf{w}_c = (0.4125, 0.5875)$. In the subsequent iterations where a new value vector is identified (Figure 3.2b-d), two new corner weights are produced. For two objectives, there are always (at most) two new corner weights per iteration, and two value vectors involved in each corner weights. For higher number of objectives, it is in theory possible to construct a partial CCS, $\mathcal{S}$ that has a corner weight for which all payoff vectors in $\mathcal{S}$ are in $\mathcal{V}_{rel}$, leading to very many new corner weights. In practice however, $|\mathcal{V}_{rel}|$ is typically very small, and only a few systems of linear equations need to be solved, leading to a limited number of new corner weights.

After calculating the new corner weights $W_{\mathbf{V}^\pi}$ at line 14 (in Algorithm 5), $\mathbf{V}$ is added to $\mathcal{S}$ at line 15. Cheng showed that finding the best payoff vector for each corner weight and adding it to the partial CCS, guarantees the best improvement to $\mathcal{S}$:

Theorem 3. *(Cheng, 1988) The maximum value of:*

$$\max_{\mathbf{w}, \mathbf{V} \in CCS} \min_{\mathbf{V}' \in \mathcal{S}} \mathbf{w} \cdot \mathbf{V} - \mathbf{w} \cdot \mathbf{V}',$$

i.e., the maximal improvement to $\mathcal{S}$ by adding a vector to it, is at one of the corner weights.

Theorem 3 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 (as is the case in Figure 3.2f).

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 `SolveSODP` is typically an expensive operation, testing all corner weights may not be feasible in MODPs. For example, in MO-CoGs, a common choice for `SolveSODP` is the *variable elimination* algorithm (which we discuss in detail in the next chapter), whose runtime can be exponential in the size of the problem. 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 to $V^*_S(\mathbf{w})$ that can be made by adding a single new value vector. 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 $\mathcal{S}$ is the

current partial CCS and $\mathcal{W}$ is the set of weights already tested with `SolveSODP`. A key advantage of OLS over Cheng's linear support is thus that these priorities can be computed without calling `SolveSODP`, obviating the need to run `SolveSODP` on all corner weights.

Definition 20. *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 $\mathcal{S}$ at the weights in $\mathcal{W}$.*

In Figure 3.2a the $\overline{CCS} = \{(16.3, 11.8), (11.6, 15.1), (16.3, 15.1)\}$. $\overline{CCS}$ is a superset of $\mathcal{S}$ and the value of $V^*_{\overline{CCS}}(\mathbf{w})$ (indicated by the dotted line) is the same as $V^*_{\mathcal{S}}(\mathbf{w})$ at all the weights in $\mathcal{W}$. For a given $\mathbf{w}$, $\texttt{maxValueLP}(\mathbf{w}, \mathcal{S}, \mathcal{W})$ finds the scalarized value of $V^*_{\overline{CCS}}(\mathbf{w})$ by solving the following *linear program (LP)*:

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

where $\mathbf{V}^*_{\mathcal{S},\mathcal{W}}$ is a vector containing $V^*_{\mathcal{S}}(\mathbf{w}')$ for all $\mathbf{w}' \in \mathcal{W}$, and $\mathbf{v}$ is a vector of variables of length d. Note that we abuse the notation $\mathcal{W}$, which in this case is a matrix whose rows correspond to all the weight vectors in the set $\mathcal{W}$.[4]

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

$$\Delta(\mathbf{w}) = V^*_{\overline{CCS}}(\mathbf{w}) - V^*_{\mathcal{S}}(\mathbf{w}).$$

Figure 3.2a shows $\Delta(\mathbf{w}_c)$ with a dashed line. We use the *maximal relative possible improvement*, $\Delta_r(\mathbf{w}) = \Delta(\mathbf{w})/V^*_{\overline{CCS}}(\mathbf{w})$, as the priority of each new corner weight $\mathbf{w} \in W_{\mathbf{V}}$. In Figure 3.2a, $\Delta_r(\mathbf{w}_c)$=$\frac{(0.4125,0.5875)\cdot((16.3,15.1)-(11.6,15.1))}{13.65625}$ = 0.141968. When a corner weight $\mathbf{w}$ is identified (line 14), it is added to Q with priority $\Delta_r(\mathbf{w})$ as long as $\Delta_r(\mathbf{w}) > \varepsilon$ (lines 17-19). In Figure 3.2 the corner weights in Q are indicated by red vertical line segments.

After $\mathbf{w}_c$ in Figure 3.2a is added to Q, it is popped off again (as it is the only element of Q). $\texttt{SolveSODP}(\mathbf{w}_c)$ generates a new value vector $(13.9, 14.3)$, yielding $\mathcal{S} = \{(16.3, 11.8), (11.6, 15.1), (13.9, 14.3)\}$, as illustrated in Figure 3.2b. The new corner weights are the points at which $(13.9, 14.3)$ intersects with $(16.3, 11.8)$ and $(11.6, 15.1)$. Testing these weights, as illustrated in Figure 3.2cd, results in 2 new payoff vectors, and two new corner weights each. After these two value vectors however, checking the weight with the highest priority in Q does not result in a new vector, reducing the maximal possible improvement at that weight to 0 (Figure 3.2e). Checking the remaining 3 corner weights in Q, also does not lead to new value vectors, causing OLS to terminate. Because the maximal improvement at these corner weights is 0 upon termination, $S = CCS$ due to Theorem 3. OLS called `SolveSODP` for only 9 weights resulting exactly in the 5 payoff vectors of the CCS. The other 7 payoff vectors in $\mathcal{V}$ (the black points of Figure 3.1) were never generated.

[4] Our implementation of OLS reduces the size of the LP by using only the subset of weights in $\mathcal{W}$ for which the policies involved in $\mathbf{w}$, $\Pi_{\mathcal{S}}(\mathbf{w})$, have been found to be optimal. This can lead to a slight overestimation of $V^*_{\overline{CCS}}(\mathbf{w})$.

3.4 Analysis

We now analyze the computational and space complexity of OLS. Because OLS is a generic algorithm that takes a single-objective solver (and possibly a policy evaluation algorithm) as a subroutine, we provide the complexity bounds of OLS in terms of the runtime and space complexities of these subroutines. We denote the runtime of a single-objective solver as R_{so} and the runtime of policy evaluation as R_{pe}, and the corresponding memory as M_{so} and M_{pe}.

Theorem 4. *The runtime of OLS is*

$$O(\ (|\varepsilon\text{–}CCS| + |\mathcal{W}_{\varepsilon-CCS}|)(R_{so} + R_{pe} + R_{nw} + R_{heur})\ ),$$

where $|\varepsilon\text{–}CCS|$ is the size of the ε-CCS (Definition 13) outputted by OLS, $|\mathcal{W}_{\varepsilon-CCS}|$ is the number of corner weights of the scalarized $V^*_{\varepsilon-CCS}(\mathbf{w})$ corresponding to the output ε-CCS, R_{nw} the time it takes to run `newCornerWeights`*, and R_{heur} the time it takes to compute the value of the optimistic CCS using* `maxValueLP`.

Proof. The runtime of one iteration of OLS is the cost of running the single-objective solver plus policy evaluation, $R_{so} + R_{pe}$, plus the overhead per corner weight $R_{nw} + R_{heur}$, multiplied by the number iterations. To count the number of iterations, we consider two cases: calls to the single-objective solver that result in adding a new vector to the partial CCS, $\mathcal{S}$ and those that do not result in a new vector but instead confirm the optimality of the scalarized value $V^*_{\mathcal{S}}(\mathbf{w})$ at that weight. The former is the size of the output of OLS, i.e., $|\varepsilon\text{–}CCS|$, while the latter is at most the number of corner weights of the scalarized value function of that same output set, $|\mathcal{W}_{\varepsilon-CCS}|$. □

Note that we can often adapt the implementation of `SolveSODP` to return the value vector of the optimal policy for a weight, alongside this optimal policy, without an increase in the complexity bounds of the single-objective solver. In this case R_{pe} becomes 0.

We have experienced that the overhead of OLS itself, i.e., computing new corner weights, R_{nw}, and calculating the maximal relative improvement, R_{heur}, is very small compared to the `SolveSODP` for the decision problems that we study in this dissertation. In practice, `newCornerWeights`($\mathbf{u}$, W_{del}, $\mathcal{S}$) computes the solutions to only a small set of linear equations (of d equations each). `maxValueLP`($\mathbf{w}$, $\mathcal{S}$, $\mathcal{W}$) computes the solutions to linear programs, which is polynomial in the size of its inputs.[5]

If $\varepsilon \neq 0$, OLS does in practice not test all the corner weights of the polyhedron spanned by the ε-CCS it outputs, as OLS only tests a corner weight if the maximal possible improvement is larger than ε. However, this cannot be guaranteed in general. Note that if $\varepsilon = 0$, OLS outputs an exact CCS, and needs to check every corner weight in order to establish that it is in fact an exact CCS.

[5]When the reduction in Footnote 4 is used, only a very small subset of $\mathcal{W}$ is used, making it even smaller.

For $d = 2$, the number of corner weights is smaller than the size, $|\varepsilon\text{–}CCS|$, of the output of OLS. Therefore, the runtime of OLS is $O(|\varepsilon\text{–}CCS|(R_{so} + R_{pe} + R_{nw} + R_{heur}))$. For $d = 3$, the number of corner weights is $2|\varepsilon\text{–}CCS|$ (minus a constant) because, when `SolveSODP` finds a new payoff vector, one corner weight is removed and three new corner weights are added. For $d = 3$, the computational complexity is thus still only $O(|\varepsilon\text{–}CCS|(R_{so} + R_{pe} + R_{nw} + R_{heur}))$. For $d > 3$, a loose bound on $|\mathcal{W}_{\varepsilon\text{–}CCS}|$ is the total number of possible combinations of d payoff vectors or boundaries: $O(\binom{|\varepsilon\text{–}CCS|+d}{d})$. 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 and Pfetsch, 2003).

Theorem 5. *(Avis and Devroye, 2000) For arbitrary d, $|\mathcal{W}_{\varepsilon\text{–}CCS}|$ is bounded by*

$$O(\binom{|\varepsilon\text{–}CCS| - \lfloor \frac{d+1}{2} \rfloor}{|\varepsilon\text{–}CCS| - d} + \binom{|\varepsilon\text{–}CCS| - \lfloor \frac{d+2}{2} \rfloor}{|\varepsilon\text{–}CCS| - d}).$$

Proof. This result follows directly from *McMullen's upper bound theorem* for facet enumeration (McMullen, 1970; Henk et al., 1997). □

Theorem 6. *The space complexity of OLS is*

$$O(d|\varepsilon\text{–}CCS| + d|\mathcal{W}_{\varepsilon\text{–}CCS}| + M_{so} + M_{pe}).$$

Proof. OLS needs to store every corner weight (a vector of length d) in the queue, which is at most $|\mathcal{W}_{\varepsilon\text{–}CCS}|$. OLS also needs to store every vector in $\mathcal{S}$ (also vectors of length d). Furthermore, when `SolveSODP` is called, the memory usage of this single-objective solver is added to the memory usage of the outer loop of OLS. The same holds for policy evaluation. □

Because OLS adds few memory requirements to that of the single-objective solver for small and medium numbers of objectives, OLS is almost as memory efficient as the single-objective solver itself in these cases. This is a big difference with inner loop methods, which need to retain sets of value vectors and partial policies, everywhere that the single-objective method would have a single value and a single partial policy.

3.5 Approximate Single-Objective Solvers

Up until now, we have assumed that OLS takes an arbitrary *exact* single-objective solver as a subroutine. In this section, we show that OLS can also be applied when the single-objective subroutine is approximate. Furthermore, when the subroutine is produces bounded approximations of at most ε error, OLS is guaranteed to produce an ε-CCS.

First, we define what it means to have an ε-approximate algorithm as a single-objective subroutine.

Algorithm 7: AOLS$(m,$ approxSolveSODP$, \varepsilon)$

Input: An MODP m, an approximate SODP subroutine ApproxSolveSODP, and max. allowed error, ε.

$\mathcal{S} \leftarrow \emptyset$; // a partial CCS
$\mathcal{W} \leftarrow \emptyset$; // set of tuples of visited weights and upper bounds, $\bar{V}_{\mathbf{w}}$
$Q \leftarrow$ an empty priority queue
foreach *extremum of the weight simplex* $\mathbf{w}_e$ **do**
  Q.add($\mathbf{w}_e, \infty$) ; // add the extrema to Q with infinite priority
end
while $\neg$*Q.isEmpty()* $\wedge$ $\neg$*timeOut* **do**
  $\mathbf{w} \leftarrow$ Q.pop()
  $\mathbf{V}^\pi, \bar{V}_{\mathbf{w}} \leftarrow$ approxSolveSODP$(m, \mathbf{w})$
  $\mathcal{W} \leftarrow \mathcal{W} \cup \{(\mathbf{w}, \bar{V}_{\mathbf{w}})\}$
  if $\mathbf{V}^\pi \notin \mathcal{S}$ **then**
    $W_{del} \leftarrow$ remove the corner weights made obsolete by $\mathbf{V}$ from Q, and store them
    $W_{del} \leftarrow \{\mathbf{w}\} \cup W_{del}$; // corner weights to remove
    $W_{\mathbf{V}^\pi} \leftarrow$ newCornerWeights$(\mathbf{V}^\pi, W_{del}, \mathcal{S})$
    remove vectors from $\mathcal{S}$ that are no longer optimal for any $\mathbf{w}$ when $\mathbf{V}^\pi$ is added
    $\mathcal{S} \leftarrow (S \cup \{\mathbf{V}^\pi\})$
    foreach $\mathbf{w} \in W_{\mathbf{V}^\pi}$ **do**
      $\Delta_r(\mathbf{w}) \leftarrow$ calculate improvement using maxValueLP$(\mathbf{w}, \mathcal{S}, \mathcal{W})$
      if $\Delta_r(\mathbf{w}) > \varepsilon$ **then**
        Q.add($\mathbf{w}$, $\Delta_r(\mathbf{w})$)
      end
    end
  end
end
return $\mathcal{S}$ *and the highest* $\Delta_r(\mathbf{w})$ *left in* Q

Definition 21. *An* ε-approximate SODP solver *is an algorithm that produces a policy whose value is at least* $(1-\varepsilon)V^*$*, where* V^* *is the optimal value for the SODP and* $\varepsilon \geq 0$.

Given an ε-approximate SODP solver we can compute a set of policies for which the scalarized value for each possible $\mathbf{w}$ is at least $1-\varepsilon$ times the optimal scalarized value, i.e., an ε-CCS (Definition 13). To this end we change the OLS algorithm in order to handle approximate solvers, leading to the *approximate OLS (AOLS)* algorithm (Algorithm 7). The differences with OLS (Algorithm 5) are highlighted in blue. We go through the differences one by one.

Firstly, in Algorithm 7 we assume that the single-objective subroutine — in this algorithm referred to as approxSolveSODP — returns both the value vector and policy[6],

[6]We assume that the value $\mathbf{V}^\pi$ is the correct value vector of π, i.e., the evaluation of the policy value is exact.

$\mathbf{V}^\pi$, and an upper bound on the scalarized value at the weight $\mathbf{w}$ for which it is called, $\bar{V}_\mathbf{w}$ (line 9). This upper bound is stored alongside $\mathbf{w}$ (lines 2 and 10), and used instead of the scalarized values in `maxValueLP`. Specifically, for a given $\mathbf{w}$, `maxValueLP` finds (line 18) the scalarized value of $V^*_{\overline{CCS}}(\mathbf{w})$ by solving:

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

where for each tuple $(\mathbf{w}, \bar{V}_\mathbf{w}) \in \mathcal{W}$, there is a row in the matrix $\mathcal{W}'$ corresponding to $\mathbf{w}$ with a corresponding element $\bar{V}_\mathbf{w}$ in the vector $\bar{\mathbf{V}}^*_{\mathcal{S},\mathcal{W}}$.

Secondly, because we have an approximate single-objective subroutine, the vectors in the partial CCS $\mathcal{S}$, may not be optimal for any $\mathbf{w}$ anymore when we add a new value vector. Therefore, we first have to remove these vectors from $\mathcal{S}$ (line 15). We can do this efficiently, by checking whether $\mathbf{V}^\pi$ is better than a vector $\mathbf{V}' \in \mathcal{S}$ for all the corner weights of $V^*_\mathcal{S}(\mathbf{w})$ for which $\mathbf{V}'$ is optimal, as this bounds the area for which $\mathbf{V}'$ is optimal w.r.t. to the other value vectors currently in $\mathcal{S}$.

Correctness

We now establish the correctness of Algorithm 7, i.e., OLS with approximate single-objective subroutines. Because the scalarized value of $V^*_{\overline{CCS}}(\mathbf{w})$ (as computed by `maxValueLP`) obtained through using the ε of the approximate `solveMDP` is no longer identical to $V^*_\mathcal{S}(\mathbf{w})$, we need to make an adjustment to Cheng's theorem.

Theorem 7. *There is a corner weight of $V^*_\mathcal{S}(\mathbf{w})$ that maximizes:*

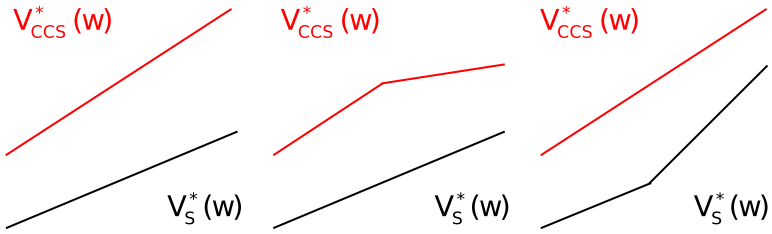
$$\Delta(\mathbf{w}) = V^*_{CCS}(\mathbf{w}) - V^*_S(\mathbf{w}),$$

where S is an intermediate set of value vectors computed by AOLS (Algorithm 7).

This theorem is identical to Cheng's, but, because $\mathcal{S}$ is no longer a subset of the CCS, the proof is different.

Proof. $\Delta(\mathbf{w})$ is the difference between two PWLC functions: $V^*_{CCS}(\mathbf{w})$ and $V^*_\mathcal{S}(\mathbf{w})$. To maximize this function, we have three possible cases, shown in Figure 3.3: (a) the maximum is at a weight that is neither a corner point of $V^*_{CCS}(\mathbf{w})$, nor of $V^*_S(\mathbf{w})$; (b) it is at a corner point of $V^*_{CCS}(\mathbf{w})$ but not of $V^*_S(\mathbf{w})$; or (c) it is at a corner point of $V^*_S(\mathbf{w})$.

Case (a) can only apply if the slope of $\Delta(\mathbf{w})$ at $\mathbf{w}$ is 0, if this is so, then the value of $\Delta(\mathbf{w})$ is equal to the value at the corner points where this slope changes. Case (b) can never occur: if $\mathbf{w}$ is a corner point of $V^*_{CCS}(\mathbf{w})$, and not of $V^*_\mathcal{S}(\mathbf{w})$, and $\Delta(\mathbf{w}) = V^*_{CCS}(\mathbf{w}) - V^*_\mathcal{S}(\mathbf{w})$ is at a maximum, then because $V^*_\mathcal{S}(\mathbf{w})$ does not change slope in $\mathbf{w}$, the change in slope for $V^*_{CCS}(\mathbf{w})$ must be negative. However, because we know that $V^*_{CCS}(\mathbf{w})$ is a PWLC function, this leads to a contradiction. Therefore, only case (c) remains. □



(a) (b) (c)

Figure 3.3: Possible cases for the maximum possible improvement of $\mathcal{S}$ with respect to the CCS.

Because the maximum possible improvement is still at the corner points of $\mathcal{S}$, even though $\mathcal{S}$ now contains ε-approximate solutions, the original scheme of calling a single-objective subroutine for the corner weights still applies.

The approximate-subroutine version of OLS terminates when the maximal possible improvement is less than or equal to $\varepsilon \cdot V^*_{\overline{CCS}}(\mathbf{w})$. In other words, when there are no corner weights with a possible improvement higher than the input slack ε.

Theorem 8. *AOLS — as specified in Algorithm 7 — terminates after a finite number of calls to an ε-approximate SODP solver* `approxSolveSODP` *and produces an ε-CCS.*

Proof. The algorithm runs until there are no corner points left in the priority queue to check, and returns $\mathcal{S}$ and the highest priority left in Q. Once a corner point is evaluated, it is never considered again because the established value lies within $(1-\varepsilon)V^*_{\overline{CCS}}(\mathbf{w})$ (as guaranteed by the ε-bound of `approxSolveSODP`). AOLS thus terminates after checking a finite number of corner weights. All other corner weights have a possible improvement less than or equal to $\varepsilon V^*_{\overline{CCS}}(\mathbf{w})$. Therefore $\mathcal{S}$ must be an ε-CCS. □

An equivalent, but subtly different version of the approximate OLS algorithm and the corresponding correctness theorem, can be established accordingly, when the single-objective subroutine provides an upper and lower bound (implying an ε), but it cannot be know beforehand how strict these bounds will be:

Corollary 2. *When an approximate single-objective solver produces a bounded approximate solution for each scalarized problem, with an error bound of at most ε, AOLS produces an ε-CCS.*

Proof. The proof is the same as for 8, by assuming that there is some ε for which `approxSolveSODP` is ε-approximate, and establishing this on the fly, by inspecting the difference between the upper and lower bounds found by the ε-approximate solver. □

Note that the runtime and memory requirements, as established in Theorems 4 and 6, are not affected by the usage of an approximate single-objective subroutine instead of an exact one, in any way other than changing the value of R_{so}, and M_{so}.

3.6 Value reuse

One possible issue with using OLS in practice is that for every corner weight $\mathbf{w}$, the single-objective subroutine is called. When there are many corner weights, this may take a long time, especially when the single-objective solver need to start from scratch. However, in many cases, we might be able to reuse part of the work done in earlier iterations of OLS, in order to *hot-start* the single-objective subroutine for a new $\mathbf{w}$.

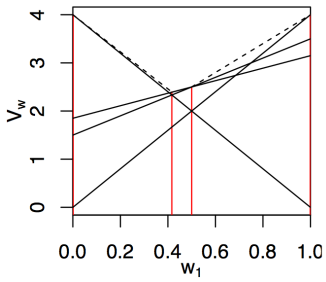


Figure 3.4: Close corner weights lead to close value vectors.

The key insight behind *reuse in OLS*, is that when two corner weights, $\mathbf{w}$ and $\mathbf{w}'$, are similar, the value vectors for these weights found by the single-objective solver are also likely to be similar, (e.g., in Figure 3.4). Therefore, we also expect that similar policies are optimal for $\mathbf{w}'$. If that is indeed the case, and we have a single-objective subroutine that can start from a previous solution — or part of this previous solution — and gradually improve, this may save us a lot of runtime.

In Algorithm 8, we show how this could be implemented. The differences with Algorithm 7 are shown in blue. We aim to show the most general form of the algorithm, and therefore use an abstract data type $I_{\mathbf{w}}$ that represents *any possible information* from a previous call to `approxSolveSODP` for a weight $\mathbf{w}$, that we may reuse in subsequent iterations. Which information can be reused depends on the specifics of the MODP, e.g., the full state-value function of a MO(PO)MDP (Chapter 5), or a reparameterization used by variational methods for coordination graphs (Chapter 4). We assume that this information is produced by the single-objective solver (line 10), and is stored together with the previous search weights (lines 2 and 11). At every iteration, all possibly reusable information from all previous iterations, I, is retrieved (line 9) and used to hot-start the single-objective subroutine on line 10.

Without further assumptions, *reuse* is a heuristic, i.e., it may improve the runtime in practice, but it does not show up in the complexity results for the algorithm. However, if we can assume that I contains enough information for `approxSolveSODP` to check that $\mathcal{S}$ is already sufficient for the new weight $\mathbf{w}$, and that this check can be performed in $R_{confirm}$, we do not require a full run of the single-objective subroutine for corner weights for which we will not find a new value vector, and can reduce the theoretical runtime guarantees.

Theorem 9. *The runtime of OLS with reuse is*

$$O(\ (|\varepsilon\text{–}CCS|)(R_{so} + R_{pe} + R_{nw} + R_{heur}) + |\mathcal{W}_{\varepsilon-CCS}|R_{confirm}\ ),$$

where $R_{confirm}$ is the time it takes to check the sufficiency of the solutions currently in $\mathcal{W}$ and their value vectors $\mathcal{S}$.

Algorithm 8: OLS+R(m, approxSolveSODP, ε)

Input: An MODP m, an approximate SODP subroutine ApproxSolveSODP, and max. allowed error, ε.

$\mathcal{S} \leftarrow \emptyset$; // a partial CCS
$\mathcal{W} \leftarrow \emptyset$; // a set of tuples of $\mathbf{w}$, $\bar{V}_{\mathbf{w}}$ and $I_{\mathbf{w}}$
$Q \leftarrow$ an empty priority queue;
foreach *extremum of the weight simplex* $\mathbf{w}_e$ **do**
  Q.add($\mathbf{w}_e, \infty$) ; // add the extrema to Q with infinite priority
end
while $\neg$*Q.isEmpty()* $\wedge$ $\neg$*timeOut* **do**
  $\mathbf{w} \leftarrow$ Q.pop();
  $I \leftarrow$ Retrieve the reusable information for relevant previous iterations from $\mathcal{W}$;
  $\mathbf{V}^\pi, \bar{V}_{\mathbf{w}}, I_{\mathbf{w}} \leftarrow$ approxSolveSODP$(m, \mathbf{w}, I)$;
  $\mathcal{W} \leftarrow \mathcal{W} \cup \{(\mathbf{w}, \bar{V}_{\mathbf{w}}, I_{\mathbf{w}})\}$;
  if $\mathbf{V}^\pi \notin \mathcal{S}$ **then**
    $W_{del} \leftarrow$ remove the corner weights made obsolete by $\mathbf{V}$ from Q, and store them
    $W_{del} \leftarrow \{\mathbf{w}\} \cup W_{del}$; // corner weights to remove
    $W_{\mathbf{V}^\pi} \leftarrow$ newCornerWeights$(\mathbf{V}^\pi, W_{del}, \mathcal{S})$;
    remove vectors from $\mathcal{S}$ that are no longer optimal for any $\mathbf{w}$ when $\mathbf{V}^\pi$ is added;
    $\mathcal{S} \leftarrow (\mathcal{S} \cup \{\mathbf{V}^\pi\})$;
    foreach $\mathbf{w} \in W_{\mathbf{V}^\pi}$ **do**
      $\Delta_r(\mathbf{w}) \leftarrow$ calculate improvement using maxValueLP$(\mathbf{w}, \mathcal{S}, \mathcal{W})$;
      if $\Delta_r(\mathbf{w}) > \varepsilon$ **then**
        Q.add($\mathbf{w}$, $\Delta_r(\mathbf{w})$);
      end
    end
  end
end
return $\mathcal{S}$ *and the highest* $\Delta_r(\mathbf{w})$ *left in* Q

This is especially important when the number of objectives is high, because — as we have seen in Theorem 5 — $|\mathcal{W}_{\varepsilon-CCS}|$ increases exponentially with the number of objectives. Note however, that the greater the amount of information stored per $\mathbf{w}$, the more memory per corner weight OLS with reuse uses. We discuss problem-specific instances of this algorithm, implementing reuse for the various problems we discuss in this dissertation in the following chapters.

Chapter 4

Coordination

In cooperative multi-agent settings agents must coordinate their behavior in order to optimize their common team payoff. Key to making coordination between agents efficient is exploiting the *loose couplings* common to such tasks: each agent's actions directly affect only a subset of the other agents. *Multi-objective coordination graphs (MO-CoGs)* (Definition 27) express such independence in a graphical model for single-shot decisions. In this chapter, we propose new methods for MO-CoGs that compute the *convex coverage set (CCS)*.

The single-objective version of MO-CoGs, i.e., *coordination graphs (CoGs)*, is well-studied, and many methods to exploit loose couplings exist, including *variable elimination* (Rosenthal, 1977; Dechter, 1998; Guestrin et al., 2002; Kok and Vlassis, 2006), *AND/OR tree search* (Dechter and Mateescu, 2007; Marinescu, 2008; Mateescu and Dechter, 2005; Yeoh et al., 2010), and *variational methods* (Wainwright and Jordan, 2008; Liu and Ihler, 2011; Sontag et al., 2011; Ihler et al., 2012). For MO-CoGs, several methods (Delle Fave et al., 2011; Dubus et al., 2009a; Marinescu et al., 2012) that have previously been developed build upon a single-objective method using an inner loop approach. These methods typically compute a *Pareto coverage set (PCS)*. For instance, Rollón (2008) introduces an inner-loop algorithm that we refer to as *multi-objective variable elimination* (MOVE), which builds upon the variable elimination algorithm and solves multi-objective coordination graphs by iteratively solving local problems to eliminate agents from the graph. In this dissertation (i.e., Section 2.3) however, we argue that the PCS is often not the most appropriate solution concept. Therefore, we propose novel algorithms that compute the CCS rather than the PCS. We define both inner and outer loop methods, and compare them to each other.

Before we can propose multi-objective methods however, we must first provide background on the single-objective methods upon which these methods build, as well as single-objective coordination graphs, in Section 4.1. Then, in Section 4.2, we define multi-objective coordination graphs formally.

In Section 4.3 we propose novel inner loop methods that compute a CCS for MO-CoGs. We propose *convex multi-objective variable elimination (CMOVE)* in Section

4.3.1, that builds upon variable elimination, and *convex tree-search (CTS)* in Section 4.3.3 that builds upon AND/OR tree search. Both CMOVE and CTS are exact methods, but while CMOVE is fast and uses a lot of memory, CTS focusses on situations in which limited memory is available.

In Section 4.4, we propose our novel outer loop methods. Firstly, we propose *variable elimination linear support (VELS)* that combines our outer loop approach — OLS (Chapter 3) — with variable elimination, and *tree search linear support (TSLS)* that combines AND/OR tree search with OLS. Because VELS and TSLS are based on the same single-objective methods as CMOVE respectively CTS, comparing these methods provides a fair comparison between outer and inner loop methods.

Finally, we introduce *variational optimistic linear support (VOLS)* that combines OLS with variational methods. For VOLS, it is not clear how to create a corresponding inner loop method, as variational methods do not have explicit maximization and summation as their core operators. However, VOLS is an important addition to the aforementioned outer loop methods; it is not exact — though it provides a bounded approximation — but makes up for this lack of optimality with very favorable empirical runtime results.

The results of comparing our algorithms to PCS methods indicate that computing the CCS is typically much less computationally intensive than computing a PCS. Furthermore, the results indicate that the outer loop approach is typically much faster than the inner loop approach for small and medium numbers of objectives. We conclude with a summary of the available trade-offs that our novel algorithms pose with respect to runtime, memory-usage, and quality of the output set in Section 4.5.

4.1 Coordination Graphs

Before going into MO-CoGs, we first provide background on the corresponding single-objective problem, i.e., *coordination graphs* (CoGs) (Guestrin et al., 2002; Kok and Vlassis, 2004), and methods for solving them. Our — and several existing — methods for solving MO-CoGs build upon these single-objective methods.

In the context of coordination graphs, what we called reward for bandits is typically called *payoff* in the literature. Payoff is usually denoted u (for *utility*). We adopt this terminology in this dissertation when talking about coordination graphs.

Definition 22. *A* coordination graph *(CoG) (Guestrin et al., 2002; Kok and 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*

- $\mathcal{U} = \{u^1, ..., u^\rho\}$ *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:* $u(\mathbf{a}) = \sum_{e=1}^{\rho} u^e(\mathbf{a}_e)$.

In order to ensure that the coordination graph is *fully cooperative*, 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) (Figure 4.1); 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.[1]

The main challenge in a CoG is that the size of the joint action space, $\mathcal{A}$, grows exponentially with the number of agents. It will thus quickly become intractable to enumerate all joint actions and their associated payoffs — which would be equivalent to flattening a CoG to a *multi-armed bandit problem (BP)* (Definition 14). Key to solving CoGs is therefore to exploit *loose couplings* between agents, i.e., each agent's behavior directly affects only a subset of the other agents.

Figure 4.1 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 4.1. We use this CoG as a running example throughout this section.

The policies for a CoG represent which *joint action* to take. A deterministic policy is a single joint action, and a stochastic policy is a probability distribution over joint actions, $\mathcal{A} \to [0, 1]$. Note that because coordination between agents is essential, we typically cannot restrict ourselves to independent distributions over local actions per agent, as such *independent policies* cannot express that certain joint actions that can result from drawing actions from these distributions should always be preferred to others (that can result from the same distributions).

For a CoG, as for a BP, deterministic policies suffice.

[1] In the literature, many different names are in use for the CoG model, as it has applications to many different problem domains. The *constraint optimization problem (COP)* (Yeoh et al., 2010) is identical to a CoG. Other names in use for CoGs and COPs are *weighted constraint satisfaction problems (WCSPs)* (Rollón, 2008) and we have used *collaborative graphical games (CoGG)* (Roijers et al., 2013c) in the past ourselves. Another equivalent model is the *generalized additive independence (GAI)* network (Gonzales and Perny, 2004). (Although the graphical illustrations in GAI network papers are different from those used in CoG literature — a bipartite graph with two different types of vertices: groups of agents that participate in a local payoff function, and agents that form the overlap between two local payoff functions — the (conditional independence) structure of the payoff functions is the same.) Furthermore, the *maximum a posteriori (MAP)* assignment problem in probabilistic inference in graphical models (Pearl, 1988) is an equivalent problem.

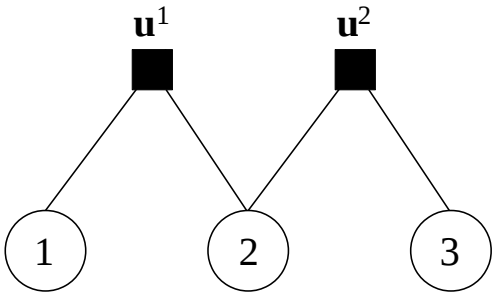


Figure 4.1: A CoG with 3 agents and 2 local payoff functions. The factor graph illustrates the loose couplings that result from the decomposition into local payoff functions. In particular, each agent's choice of action directly depends only on those of its immediate neighbors, e.g., once agent 1 knows agent 2's action, it can choose its own action without considering agent 3.

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	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 4.1: 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 ($\bar{a}_1$).

Theorem 10. *For a CoG, there is always a deterministic policy that maximizes the value.*

Proof. We can a flatten CoG to a (very large) BP (Definition 14), i.e., we can see the team of cooperative agents as a single centralized agent with a very large action space (i.e., $\mathcal{A}$), with associated payoffs $u(\mathbf{a})$, and we know that for a BP there is always a deterministic policy that maximizes the value. □

Even though we can in theory flatten a CoG to a BP, this is typically not possible in practice as the size of the action space is exponential in the number of agents, and therefore becomes too large to enumerate for all but small numbers of agents. Therefore we need special solution methods that exploit the structure of the reward function.There are many algorithms that do this; these can roughly be subdivided into four major classes of algorithms: message passing (Bishop, 2006; Kok and Vlassis, 2006; Vlassis et al., 2004), variable elimination (also called non-serial dynamic programming and bucket elimination) (Dechter, 1998; Rosenthal, 1977), search-based methods (Dechter and Mateescu, 2007; Furcy and Koenig, 2005; Marinescu, 2008), and variational methods (Liu and Ihler, 2013; Sontag et al., 2011).

In the following subsections, we summarize the three methods that we use as a starting point to create new methods for MO-CoGs, and on which several existing multi-objective algorithms are already based. Because message-passing algorithms

typically do not have quality guarantees on the joint actions they produce; we focus on the variable elimination algorithm, which is fast and produces optimal joint actions; one subclass of search-based algorithms called *AND/OR tree search*, which can produce the optimal joint actions with limited memory-usage (Mateescu and Dechter, 2005); and variational methods, which are fast, but do not always produce optimal joint actions. However, variational methods typically do return a bound on how much value is lost when they do not produce the optimal joint action. Therefore, we can use these methods as a basis for creating multi-objective methods that produce ε-CSs.

4.1.1 Variable Elimination

We first discuss the *variable elimination (VE)* algorithm, on which several multi-objective extensions (e.g., Rollón and Larrosa (2006); Rollón (2008)) build, including our own CMOVE (Section 4.3.1) and VELS (Section 4.4.1) algorithms.

VE exploits the loose couplings expressed by the local payoff functions to efficiently compute the optimal joint action. The optimal joint action is that joint action, $\mathbf{a}$, that maximizes the team payoff, $u(\mathbf{a})$. 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 best responses are used to construct a new local payoff function that encodes the values of the best responses. The new local payoff function then 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 (inner loop) extension we present in Section 4.3.1.

VE eliminates agents from the graph in a predetermined order. This order is typically determined heuristically (e.g., following Koller and Friedman (2009)), because finding the optimal elimination order is itself NP-hard (Arnborg, 1985; Dechter, 1998).

Algorithm 9 shows pseudocode for the elimination of a single agent i. First, VE determines the set, $\mathcal{U}_i$, of local payoff functions connected to i, and the set, n_i, of neighboring agents of i (lines 1-2).

Definition 23. *The set, $\mathcal{U}_i$, of neighboring local payoff functions of i is the set of all local payoff functions that have agent i in scope.*

Definition 24. *The set, n_i, of neighboring agents of 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, VE constructs a new local payoff function, $u^{new}(\mathbf{a}_{n_i})$, 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

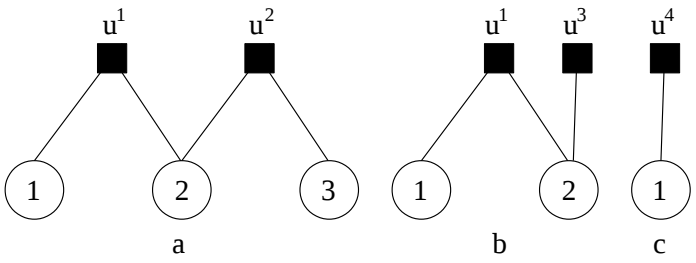


Figure 4.2: (a) A CoG with 3 agents and 2 local payoff functions (b) after eliminating agent 3 by adding u^3 (c) after eliminating agent 2 by adding u^4.

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 these tags. In this manner, the entire joint action is constructed incrementally. VE maintains the value of the best response by taking the maximum of these payoffs, and storing this maximal payoff in the new local payoff function, $u^{new}(\mathbf{a}_{n_i})$, on line 11. After the value for each $\mathbf{a}_{n_i}$ is computed, VE eliminates the agent and all payoff functions in $\mathcal{U}_i$ and replaces them with the newly constructed local payoff function (line 13).

Algorithm 9: `elimVE`$(\mathcal{U}, i)$

Input: A set, $\mathcal{U}$, of local payoff functions, 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 $\quad \mathcal{S} \leftarrow \emptyset$; `// set of action values for` i
6 $\quad$ **foreach** $a_i \in \mathcal{A}_i$ **do**
7 $\quad\quad v \leftarrow \sum_{u_j \in \mathcal{U}_i} u^j(\mathbf{a}_{n_i}, a_i)$
8 $\quad\quad$ tag v with a_i
9 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
10 $\quad$ **end**
11 $\quad u^{new}(\mathbf{a}_{n_i}) \leftarrow \max(\mathcal{S})$; `// pick maximal value`
12 **end**
13 **return** $(\mathcal{U} \setminus \mathcal{U}_i) \cup \{u^{new}\}$

Now, let us consider the example of Figure 4.1 and Table 4.1. The optimal payoff maximizes the sum of the two payoff functions:

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

The working of VE is illustrated in Figure 4.2, for the elimination order $[3, 2, 1]$. VE eliminates agent 3 first, by pushing 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 3:

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

which leads to the new factor graph depicted in Figure 4.1b. 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 4.1.

Because we ultimately want the optimal joint action as well as the optimal payoff, VE needs to store the actions of agent 3 that correspond to the values in the new local payoff factor. In our adaption of VE, the algorithm tags each payoff of u^3 with the action of agent 3 that generates it. We can thus think of $u^3(a_2)$ as a tuple of `value` and `tags`, where `value` is a scalar payoff, and `tags` is a list of individual agent actions. We denote such a tuple 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 4.1c:

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

VE appends the new tags for agent 2 to the existing tags for agent 3, yielding: $u^4(\dot{a}_1) = \max_{a_2} u^1(\dot{a}_1, a_2) + u^3(a_2) = (3.25)_{\dot{a}_2} + (2.5)_{\dot{a}_2\dot{a}_3} = (5.75)_{\dot{a}_2\dot{a}_3}$ 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 — $(5.75)_{\dot{a}_1\dot{a}_2\dot{a}_3}$ — with the optimal action contained in the tags.

The computational complexity of VE is exponential in the *induced width*, w,

Theorem 11. *(Guestrin et al., 2002) 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.*

The induced width is limited by the number of agents; when the factor graph is fully connected, i.e., every agent shares a local payoff function with every other agent, w is equal to n. In practice however, w is typically much smaller than n.

The space complexity of VE is also exponential in the induced width:

Theorem 12. *(Dechter, 1998) The space complexity of VE is* $O(\ n\ |\mathcal{A}_{max}|^w)$.

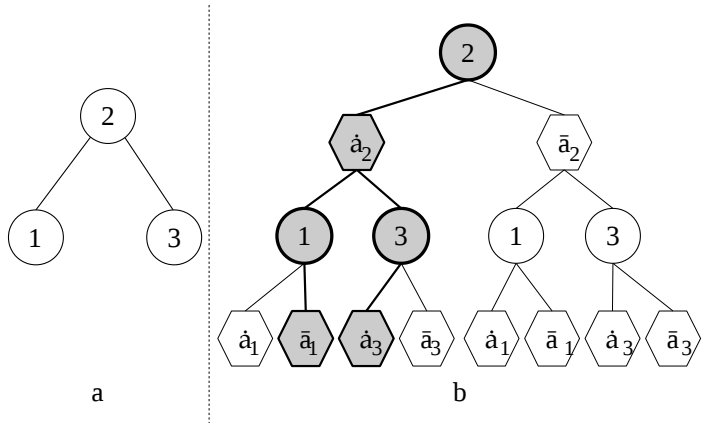


Figure 4.3: (a) a pseudo tree, (b) a corresponding AND/OR search tree.

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[2] other methods focus on memory efficiency instead (Mateescu and Dechter, 2005). We discuss a class of more memory efficient — but still exact — methods in the next section.

4.1.2 AND/OR tree search

AND/OR tree search (Dechter and Mateescu, 2007; Marinescu, 2008; Mateescu and Dechter, 2005; Yeoh et al., 2010) is a class of algorithms for solving CoGs that can be tuned to provide better space complexity guarantees than VE. However, this improvement in space complexity does come at the price of a worse runtime complexity (Mateescu and Dechter, 2005). In this section we provide a brief background on AND/OR tree search. For a broader overview of AND/OR tree search for CoGs and related models please see the work of Dechter (2013) and Marinescu (2008). For multi-objective versions of AND/OR tree search that compute a PCS, please see the work of Marinescu (2009, 2011).

The first step in an AND/OR tree search algorithm, is to convert the CoG to a *pseudo tree (PT)*. Given a PT, 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 4.3a shows the PT for the example CoG of Figure 4.1 and Table 4.1.

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

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 possible actions, $a_i \in \mathcal{A}_i$. In turn, the children of these AND-nodes are OR-nodes corresponding to agent i's children in the PT. Because each action (AND-node) of agent i has the same agents under it as OR-nodes, the agents and actions can appear in the tree multiple times. Figure 4.3b shows an AOST corresponding to the PT of Figure 4.3a.

A specific joint action can be constructed by traversing the tree, starting at the root and selecting one alternative from the children of each OR-node, i.e., one action for each agent, and continuing down all children of each AND-node. For example, in Figure 4.3b, 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.

Definition 25. *The value of an AND-node, v_{a_i}, that represents an action a_i of an agent i is the sum of the local payoff functions, $u^e(\mathbf{a}_e)$, that have i in scope (i.e., $i \in e$) for which a_i, together with its AND-node ancestors' actions, specifies a complete local joint action, $\mathbf{a}_e$.*

For example, in the AOST of Figure 4.3b, 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-node, 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 26. *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 25) 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 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 our implementation of VE, our implementation of TS employs a tagging scheme, tagging the value of a subtree with the actions that maximize it.

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

Theorem 13. *(Mateescu and Dechter, 2005) The space complexity of TS is $O(n)$, where n is the number of agents.*

Proof. Because TS performs depth-first search, creating and deleting nodes on the fly, at most $O(n)$ nodes need to exist at any point during execution of TS. □

Theorem 14. *(Mateescu and Dechter, 2005) The computational 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. □

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

Note that this theorem hold for CoGs and MO-CoGs alike, because it concerns only the structure of the graphs.

To obtain PTs in practice, we perform two steps: first, we use the same heuristic as for VE 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 15 — using the procedure proposed by Bayardo and Miranker.

Thus, combining Theorems 13, 14 and 15 shows that, when there are few agents, TS can be much more memory efficient than VE with a relatively small negative effect on the runtime.

4.1.3 Variational methods

The CoG solvers that we have described in the previous two subsections — VE and TS — are both exact methods and their computational complexity is exponential in induced width, or worse. In this section, we describe a class of algorithms called *variational algorithms*. Variational algorithms are not exact, but can provide a bounded approximate solution. In return for the loss of optimality, variational methods have runtimes that are sub-exponential in practice.

Variational algorithms (Sontag et al., 2011; Wainwright and Jordan, 2008) bound the maximal payoff of a single-objective coordination graph. We denote this upper bound as $\bar{u}$. At each iteration of a variational algorithm, this upper bound is tightened, by *reparameterizing* and possibly *restructuring* the CoG.

As an example of how upper bounds are computed, we use dual decomposition, which is a popular and illustrative variational algorithm.

Dual decomposition relaxes the full maximization into an easily evaluated bound based on local payoff functions, $u^e \in \mathcal{U}$,

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{\mathbf{a}} \sum_{u^e \in \mathcal{U}} u^e(\mathbf{a}_e) \leq \sum_{u^e \in \mathcal{U}} \max_{\mathbf{a}_e} u^e(\mathbf{a}_e) = \bar{u}. \tag{4.1}$$

Then, this bound is iteratively tightened by re-parameterizing the individual functions in $\mathcal{U}$, i.e., one finds a set of equivalent local payoffs $\mathcal{U}'$ such that the total payoff is unchanged,

$$\forall \mathbf{a} \quad u'(\mathbf{a}) = \sum_{u'^e \in \mathcal{U}'} u'^e(\mathbf{a}_e) = \sum_{u^e \in \mathcal{U}} u^e(\mathbf{a}_e) = u(\mathbf{a}),$$

while minimizing the decomposed upper bound (the right-hand side of Equation 4.1). Dual decomposition achieves the equality of $u'(\mathbf{a})$ and $u(\mathbf{a}_e)$, and the tightening of the upper bound via Lagrangian multipliers. For each original local payoff function $u^e(\mathbf{a}_e)$ with more than 1 agent in scope, dual decomposition creates a reparameterized local payoff function $\mathbf{u}'^e \in \mathcal{U}'$ that is identically scoped, and is structured as:

$$u'^e(\mathbf{a}_e) = u^e(\mathbf{a}_e) - \sum_{i \in e} \lambda_i^e(a_i),$$

where, $\lambda_i^e(a_i)$ is the Lagrangian multiplier for agent i for the local payoff function with scope e, and for each agent separately, dual decomposition creates a local payoff function, $u'^i(a_i) \in \mathcal{U}'$:

$$u'^i(a_i) = u^i(a_i) + \sum_{e \ni i} \lambda_i^e(a_i),$$

where $u^i(a_i)$ represents an original payoff function with only agent i in scope (if there is no such local payoff function in $\mathcal{U}$, we assume $u^i(a_i) = 0$), and $\sum_{e \ni i} \lambda_i^e(a_i)$ is a sum over the Lagrangian multipliers used in the larger local payoff functions $u'^e(\mathbf{a}_e)$ in which agent i participates. This ensures that the sum over all payoff functions in $\mathcal{U}'$ is identical to the sum over all payoff functions in $\mathcal{U}$ for all joint actions $\mathbf{a}$, as the $\lambda_i^e(a_i)$ components of the functions with one agent and those with more than one agent in scope cancel each other out.

After it is ensured that the total team payoff stays the same for a variational method, the remaining problem of finding equivalent sets of local payoffs $\mathcal{U}'$ while minimizing the upper bound, is a convex optimization problem and can be solved iteratively using gradient-based or fixed-point techniques (Sontag et al., 2011).

The upper bound corresponds to an optimization of the individual $u^e(\mathbf{a}_e)$. If the optimal local actions for each individual local payoff function $\mathbf{a}_e^*$ are all consistent with some global joint action $\mathbf{a}^*$, then $\mathbf{a}^*$ is also the global optimum of $u(\mathbf{a})$. In practice however, the decomposition bound may not be able to find the global optimum. Therefore, variational methods typically also assemble a joint action $\mathbf{a}_l$ that provides a lower bound

$$\underline{u} = u(\mathbf{a}_l).$$

Assembling $\mathbf{a}_l$ can for example be done by greedy assignment: for each local payoff $u'^e(\mathbf{a}_e)$, we assign the elements of $\mathbf{a}_e$ that are not already assigned in $\mathbf{a}_l$ by maximizing the local function, conditioned on the already-assigned elements.

4.2 Multi-objective Coordination Graphs

When we extend the CoG to the multi-objective setting, it becomes harder to solve. We have to not only consider stochastic policies, but a priori scalarization of the problem can become infeasible, even when we know the exact scalarization function, f, and its parameters $\mathbf{w}$.

Definition 27. *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 in a CoG but,*

- $\mathcal{U} = \{\mathbf{u}^1, ..., \mathbf{u}^\rho\}$ *is a set of* ρ*,* d*-dimensional* local reward functions.

The total team payoff is the sum of local vector-valued rewards: $\mathbf{u}(\mathbf{a}) = \sum_{e=1}^{\rho} \mathbf{u}^e(\mathbf{a}_e)$. *We use* u_i *to indicate the value of the* i*-th objective.*

Firstly, we observe that even when we know f, and its parameters $\mathbf{w}$, but f is non-linear, it is not possible to scalarize the MO-CoG a priori while retaining the same structure for the reward function. This is because f does not distribute over the sum,

$$f(\mathbf{u}(\mathbf{a}), \mathbf{w}) \neq \sum_{e=1}^{\rho} f(\mathbf{u}^e(\mathbf{a}_e), \mathbf{w}),$$

when f is non-linear. However, when f is linear, the scalarization is possible:

$$\mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \sum_{e=1}^{\rho} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e). \tag{4.2}$$

This observation leads directly to a conclusion about the sufficiency of deterministic policies for the CCS.

Corollary 3. *For a MO-CoG, there is always a CCS with only deterministic policies, even when stochastic policies are allowed.*

Proof. A CCS is an optimal solution set for all possible linear scalarizations. For all weights $\mathbf{w}$ in a linear scalarization function, we can translate the MO-CoG to a single-objective CoG using Equation 4.2. For the resulting CoG we know that there exists a deterministic policy that maximizes the scalarized value. Therefore, for all possible linear scalarizations there exists a deterministic policy that is optimal, and thus there is also a CCS consisting of only deterministic policies. □

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	(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 4.2: The two-dimensional payoff matrices for $\mathbf{u}^1(a_1, a_2)$ (left) and $\mathbf{u}^2(a_2, a_3)$ (right).

Secondly, we observe that allowing stochastic policies can improve the maximal scalarized value for non-linear f. This follows directly from flattening the MO-CoG to an MOBP, analogous to Theorem 10, and Observation 1. Therefore, the PCS of deterministic policies is typically not a subset of the PCS of stochastic policies. However, because Equation 4.2 fulfills the conditions of Assumption 1, we know that Theorem 1 holds, i.e., that a PCS of stochastic policies can be constructed from a CCS of deterministic policies.

Existing solution methods for MO-CoGs (Delle Fave et al., 2011; Dubus et al., 2009b; Marinescu, 2009, 2011; Rollón, 2008; Rollón and Larrosa, 2006), (as far as we are aware) all focus on finding the PCS of deterministic policies, which is axiomatically assumed to be the optimal solution set. From a utility-based perspective however, we argue that this is often not the most suitable solution set; when we focus on the CCS of deterministic policies, we have a sufficient solution set for when the scalarization function, f, is linear, and can construct a sufficient solution set from this CCS when f is non-linear, but stochastic policies are allowed.

In the following sections we propose novel algorithms for computing the CCS of deterministic policies for MO-CoGs. To illustrate the workings of these algorithms, we use an example MO-CoG with an identical structure to Figure 4.1 but with vector-valued local payoff functions. These local payoffs are given in Table 4.2.

4.3 Inner Loop CCS Methods for MO-CoGs

In this section we propose two inner loop methods for computing the CCS. These methods replace the summation and maximization operators by cross-sum and pruning operators, as described in Section 3.1.1.

In order to apply the inner loop approach, we first need to be able to work with sets of value vectors rather than single vectors. Therefore, we first translate the MO-CoG to a set of *value set factors* (VSFs), $\mathcal{F}$, instead of the set of local payoff functions $\mathcal{U}$. Each VSF, $f^e \in \mathcal{F}$, is a function mapping local joint actions, $\mathbf{a}_e$ to sets of payoff vectors. The initial VSFs are constructed from the local payoff functions, $\mathbf{u}^e \in \mathcal{U}$, such that

$$f^e(\mathbf{a}_e) = \{\mathbf{u}^e(\mathbf{a}_e)\}, \tag{4.3}$$

i.e., each VSF maps a local joint action to the *singleton set* containing only that action's local payoff.

Using these VSFs, we can now define the set of all possible (team) payoff vectors, $\mathcal{V}$ 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 $\bigoplus_{f^e \in \mathcal{F}}$ is the cross-sum (Definition 3.2) across all VSFs.

The CCS can now be calculated by applying a pruning operator `CPrune` (Algorithm 1) that removes all C-dominated vectors from a set of value vectors, to $\mathcal{V}$:

$$CCS(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\mathcal{V}(\mathcal{F})) = \texttt{CPrune}(\bigcup_{\mathbf{a}} \bigoplus_{f^e \in \mathcal{F}} f^e(\mathbf{a}_e)). \tag{4.4}$$

A naive, *non-graphical approach* to compute the CCS would simply compute the righthand side of Equation 4.4, i.e., it would compute $\mathcal{V}(\mathcal{F})$ explicitly by looping over all actions, and for each action looping over all local VSFs, and then pruning that set down to a CCS. This corresponds to *flattening* a MO-CoG to an MOBP (Definition 15)

Theorem 16. *The computational complexity of computing a CCS of a MO-CoG containing ρ local payoff functions, following the non-graphical approach (Equation 4.4) is:*

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

Proof. $\mathcal{V}$ 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`, whose runtime is $O(d|\mathcal{V}||PCS| + |PCS|P(d|CCS|))$ (Theorem 2). □

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

In the rest of this section, we show how to use VE (Section 4.1.1) and TS (Section 4.1.2) as a basis for new algorithms that compute the CCS and scale much better in the number of agents than a non-graphical approach.[3]

4.3.1 Convex Multi-Objective Variable Elimination

In MO-CoGs, we can compute a CCS much more efficiently when we exploit the MO-CoG's graphical structure. Like in the VE algorithm for CoGs, we can solve the MO-CoG as a series of local subproblems, by *eliminating* agents and manipulating the

[3]Note that for variational methods (Section 4.1.3) an inner loop approach does not apply because they use reparameterization rather than summation and maximization. Therefore, we only propose an outer loop method building on variational methods in Section 4.4.5.

set of VSFs $\mathcal{F}$ which describe the MO-CoG. We call the resulting algorithm *Convex Multi-Objective Variable Elimination (CMOVE)*. CMOVE is an extension to Rollón and Larrosa's Pareto-based extension of VE, which we refer to as PMOVE (Rollón and Larrosa, 2006).

The key idea of CMOVE 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, minimizing the number of payoff vectors that are calculated at the global level. Minimizing the number of payoff vectors that are calculated greatly reduces computation time.

Eliminating agents

First, we describe the `elim` operator for eliminating agents from a set of VSF. This operator corresponds to the `elim` operator used by VE (Algorithm 9) for eliminating agents in single-objective CoGs. We first need to update our definition of neighboring local payoff functions (Definition 23), to neighboring VSFs.

Definition 28. *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 24. For each possible local joint action of n_i, we now compute an LCCS — a local CCS — 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 C-dominated ones. This can be achieved by taking the cross-sum of all the VSFs in the local subproblem, and then pruning.

Definition 29. *If we fix all actions in $\mathbf{a}_{n_i}$, but not a_i, the set of all payoff vectors 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 29, we can now formally define the LCCS as the CCS of $\mathcal{V}_i$:

Definition 30. *A local CCS, an LCCS, is the C-undominated subset of $\mathcal{V}_i(\mathcal{F}_i, \mathbf{a}_{n_i})$:*

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

Because the $LCCS_i$ is the CCS of a specified set of vectors, we can compute the LCCS by standard pruning algorithms, such as `CPrune` (Algorithm 1). Using 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:

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

Eliminating an agent reduces the number of agents and VSFs in the graph, and forms the cornerstone operator of the CMOVE algorithm. Before defining the full CMOVE algorithm, we first prove the correctness of `elim`. Particularly (as described in Section 3.1.1), we must prove that no necessary vectors are lost when applying the `elim` operator.

We show that the maximal scalarized payoff, for any $\mathbf{w}$, cannot be lost as a result of `elim`, i.e., no necessary payoff vectors for creating a CCS are lost.

Theorem 17. `elim` *preserves the* CCS*, i.e.,*

$$\forall i \, \forall \mathcal{F} \; CCS(\mathcal{V}(\mathcal{F})) = CCS(\mathcal{V}(\texttt{elim}(\mathcal{F}, i))).$$

Proof. By definition the CCS of a MO-CoG, contains at least one payoff vector that maximizes the scalarized value for every $\mathbf{w}$. Therefore, for each vector $\mathbf{u}(\mathbf{a})$ that is optimal for at least one $\mathbf{w}$, there must be a vector that achieves the same scalarized value for that $\mathbf{w}$:

$$\begin{aligned} \forall \mathbf{w} \quad & \left(\mathbf{a} = \arg\max_{\mathbf{a} \in \mathcal{A}} \mathbf{w} \cdot \mathbf{u}(\mathbf{a})\right) \implies \\ \exists \mathbf{a}' \quad & \mathbf{u}(\mathbf{a}') \in CCS(\mathcal{V}(\mathcal{F})) \;\; \wedge \;\; \mathbf{w} \cdot \mathbf{u}(\mathbf{a}) = \mathbf{w} \cdot \mathbf{u}(\mathbf{a}'). \end{aligned} \tag{4.5}$$

If and only if this is not the case, necessary values are lost.

First, we observe that 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 (by definition). Second, we observe that 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 4.5, 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 $\mathbf{w} \cdot f^{new}(\mathbf{a}_{n_i}) = \max_{a_i} \sum_{e \in n_i} \mathbf{w} \cdot \mathbf{u}^e(\mathbf{a}_e)$ — for all $\mathbf{w}$ — 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.

Algorithm

Using `elim`, we now present the *convex multi-objective variable elimination (CMOVE)* algorithm. In our implementation `elim` uses `CPrune` to compute the local CCSs. 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.

As previously mentioned, CMOVE is an extension to Rollón and Larrosa's Pareto-based extension of VE, which we refer to as PMOVE (Rollón and 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 4.1.1.

Algorithm 10 presents an abstract version of CMOVE that leaves the pruning operators unspecified. Depending on preference, these pruning operators can be filled in with `PPrune` (Algorithm 2), or `CPrune` (Algorithm 1), or other algorithms for computing PCSs or CCSs. Depending on which pruning operators are used in which points in the algorithm, Algorithm 10 correctly computes a correct CCS or PCS.[4]

CMOVE first translates the problem into a set of *vector-set factors* (VSFs), $\mathcal{F}$ on line 1, according to Equation 4.3. Next, CMOVE iteratively eliminates agents using `elim` (line 2–5). The elimination order can be determined using techniques devised for single-objective VE (e.g., the method proposed by Koller and Friedman (2009) for this purpose).

Algorithm 11 shows our implementation of `elim`, parameterized with two pruning operators, `prune1` and `prune2`. These pruning operators correspond to two different pruning locations inside the operator that computes $LCCS_i$: $\texttt{ComputeLCCS}_\texttt{i}(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2})$.

[4] If more information is known about the scalarization function, f (Definition 1), this could be incorporated in new pruning algorithms, that could be used here as well. E.g., if it is known that objective 1 is worth as least twice as much as objective 2, inside a linear scalarization function, this could be incorporated into the pruning operators, without affecting the correctness of the algorithm. However, what is possible in this respect highly depends on what information about f is available, and is beyond the scope of this dissertation.

Algorithm 10: 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 $\quad i \leftarrow$ q.dequeue()
4 $\quad \mathcal{F} \leftarrow \texttt{elim}(\mathcal{F}, i, \texttt{prune1}, \texttt{prune2})$
5 **end**
6 $f \leftarrow$ retrieve final factor from $\mathcal{F}$
7 $\mathcal{S} \leftarrow f(a_\emptyset)$
8 **return** prune3($\mathcal{S}$)

Algorithm 11: elim($\mathcal{F}, i$, prune1, prune2)

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

1 $n_i \leftarrow$ the set of neighboring agents of i
2 $\mathcal{F}_i \leftarrow$ the subset of VSF that have i in scope
3 $f^{new}(\mathbf{a}_{n_i}) \leftarrow$ a new VSF
4 **foreach** $\mathbf{a}_{n_i} \in \mathcal{A}_{n_i}$ **do**
5 $\quad f^{new}(\mathbf{a}_{n_i}) \leftarrow \texttt{ComputeLCCS}_\texttt{i}(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2})$
6 **end**
7 $\mathcal{F} \leftarrow \mathcal{F} \setminus \mathcal{F}_i \cup \{f^{new}\}$
8 **return** $\mathcal{F}$

$\texttt{ComputeLCCS}_\texttt{i}$ 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:

$$\texttt{ComputeLCCS}_\texttt{i}(\mathcal{F}_i, \mathbf{a}_{n_i}, \texttt{prune1}, \texttt{prune2}) = \texttt{prune2}(\bigcup_{a_i} \hat{\bigoplus}_{f^e \in \mathcal{F}_i} f^e(\mathbf{a}_e)). \quad (4.6)$$

prune1 is applied to each cross-sum of two sets, via the $\hat{\oplus}$ operator, leading to *incremental pruning* (Cassandra et al., 1997), i.e.,

$$A \hat{\oplus} B \hat{\oplus} C = \texttt{prune1}(A \oplus \texttt{prune1}(B \oplus C)).$$

prune2 is applied at a coarser level, after the union. CMOVE applies elim iteratively until no agents remain, resulting in a CCS. 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. We call this pruner prune3. 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. 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`).

When either `prune2` or `prune3` (or both) compute CCSs, the CMOVE algorithms correctly computes the CCS. When no pruning takes place for `prune3` and `prune1` = `prune2` = `PPrune`, the resulting algorithm computes the PCS rather than the CCS, and is equivalent to the PMOVE algorithm proposed by Rollón and Larrosa (2006).

When comparing CMOVE to PMOVE we make the following observation: the local coverage sets are input to the next subproblem in agent elimination sequence. A smaller local coverage set thus results in a smaller next subproblem. Because it is always possible to have a CCS that is a subset of the smallest possible PCS, the subproblems in CMOVE are thus always smaller than the corresponding subproblems in PMOVE. A key insight behind CMOVE is that this results in faster computation overall for CMOVE compared to PMOVE. Furthermore, we discuss variants of CMOVE, with different instantiations of the pruning operators, that lead to different trade-offs between pruning and the size of the subproblems.

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 dissertation, we consider two variants: *Basic CMOVE*, which does not use `prune1` and `prune3` and only prunes at `prune2` using `CPrune`, and *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.[5]

Example

Consider the example in Figure 4.1a, using the payoffs defined by Table 4.2, and apply CMOVE, using `CPrune` for `prune2`, and no pruning for `prune1` and `prune3`, i.e., Basic CMOVE.

First, CMOVE creates the VSFs f^1 and f^2 from $\mathbf{u}^1$ and $\mathbf{u}^2$. 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

[5]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.

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 4.1b.

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

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

The first cross sum 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 4.1c.

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

Analysis

We now analyze the correctness and complexity of CMOVE.

Theorem 18. *CMOVE 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 17), no necessary vectors are lost. Furthermore, no excess payoff vectors are retained; 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 19. *The computational complexity of CMOVE is*

$$O(\ n\ |\mathcal{A}_{max}|^{w_a}\ (w_f\ R_1 + R_2) + R_3\ ), \tag{4.7}$$

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 4.6) 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 10). □

R_1, R_2, and R_3 — the runtime of the pruning operators — depend on the size of the local subproblems. Specifically, the runtime of `CPrune` is $O(d|\mathcal{V}||PCS| + |PCS|P(d|CCS|))$ (Theorem 2), where $|\mathcal{V}|$ is the number of vectors inputted to `CPrune`. How big these input sets are, depends on how much pruning has been done in earlier iterations and at finer levels. Specifically, 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.

CMOVE is exponential only in w_a, and not in the number of agents as in the non-graphical approach (Theorem 16). In this respect, our results are similar to those provided for PMOVE by 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. Such bounds can be very loose or even impossible to define in practice, e.g., when the payoff values are real-valued in one or more objectives. For this reason, we provide a description of the computational complexity that makes explicit the dependence on the effectiveness of pruning, i.e., in terms of R_1, R_2, and R_3. While such complexity bounds are not better in the worst case — when no pruning is possible — they do allow greater insight into the runtimes of the algorithms we evaluate.

Besides the computational complexity, another important aspect of the behavior of MO-CoGs is the space complexity. In fact, as we show empirically in Section 4.3.4, memory is often the bottleneck for solving MO-CoGs:

Theorem 20. *The space complexity of CMOVE is*

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

where $|LCCS_{max}|$ is maximum size of a local CCS, ρ is the original number of VSFs, $|e_{max}|$ is the maximum scope size of the original VSFs, and the $|\mathcal{F}_{max}|$ is the maximal number of VSFs that get replaced by a new VSF.

Proof. CMOVE computes a local CCS — using `CPrune` — 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 VSFs. Each payoff vector stores d real numbers.

The size of the input of `CPrune` depends on the number of VSFs, $|\mathcal{F}_{max}|$, that get replaced by a new VSF, and the size of the local CCSs in the VSFs that are replaced, i.e., the size of the cross-sum in Equation 4.6. Note that, in the case of incremental pruning (`prune1` = `CPrune`), the size of this cross-sum is limited even further, by pruning after each cross-sum of 2 local CCSs.

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 equivalent to Theorem 20, but with the size of a local PCS, $|LPCS_{max}|$, instead of $|LCCS_{max}|$. Because an LCCS is a subset of the corresponding LPCS[6], CMOVE is thus strictly more memory efficient than PMOVE.

Note that Theorem 20 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.

4.3.2 Experiments: CMOVE versus PMOVE

In order to test the efficiency of CMOVE, we compare basic CMOVE and incremental CMOVE to PMOVE[7] and the non-graphical approach (described at the beginning of Section 4.3) for problems with varying numbers of agents and objectives. We also analyze how these runtimes correspond to the sizes of the PCS and CCS, as well as the induced width.

Because CMOVE and PMOVE are inner loop methods based on the same schema, i.e., VE, these experiments indicate how inner loop methods that compute the CCS compare to inner loop methods that compute the CCS. Furthermore, we see how the different variants of CMOVE compare to each other.

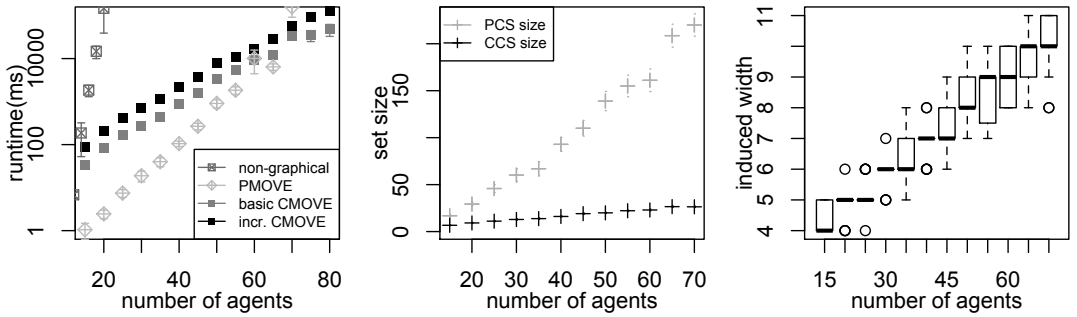
We use two types of experiments. The first experiments are done with random MO-CoGs in which we can directly control all variables. In the second experiment, we use Mining Day, a more realistic benchmark, that is more structured than random MO-CoGs but still randomized.

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 $[0, 10]$.

[6] Our implementation of `CPrune` (Algorithm 1) uses `PPrune` as a pre-processing step (on line 1).

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



(a) (b) (c)

Figure 4.4: (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 as a boxplot.

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 4.4 shows the results, averaged over 20 MO-CoGs for each number of agents. The runtime (Figure 4.4a) 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 4.4b). 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 4.4c, we see that the induced width increases linearly with the number of agents for random graphs.

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

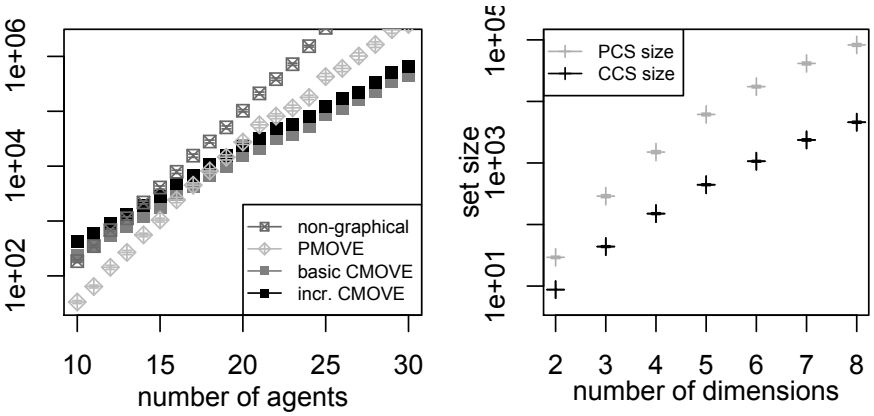


Figure 4.5: Runtimes (ms) for the non-graphical method, PMOVE and CMOVE in log-scale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of agents and 5 objectives.

computer with 24 GB memory. Figure 4.5 (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 4.5 (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 4.6 (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 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 $d = 5$,

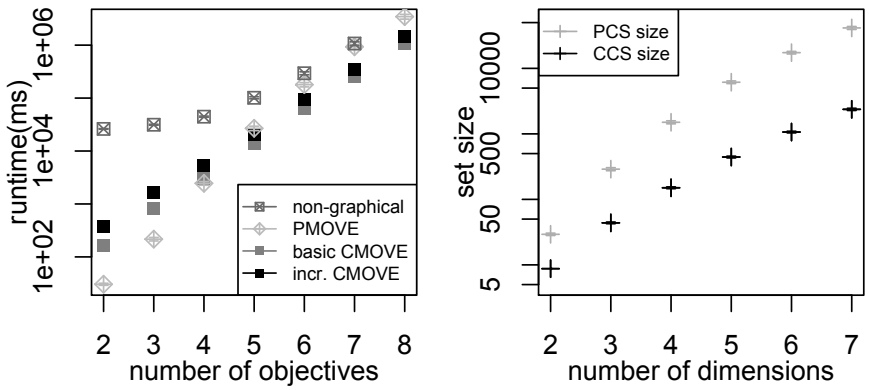


Figure 4.6: Runtimes (ms) for the non-graphical method, PMOVE and CMOVE in logscale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of objectives.

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 4.6 (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.

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 1.2). 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, the mining company wants to use the latest possible price information, and not lose time recomputing the optimal strategy

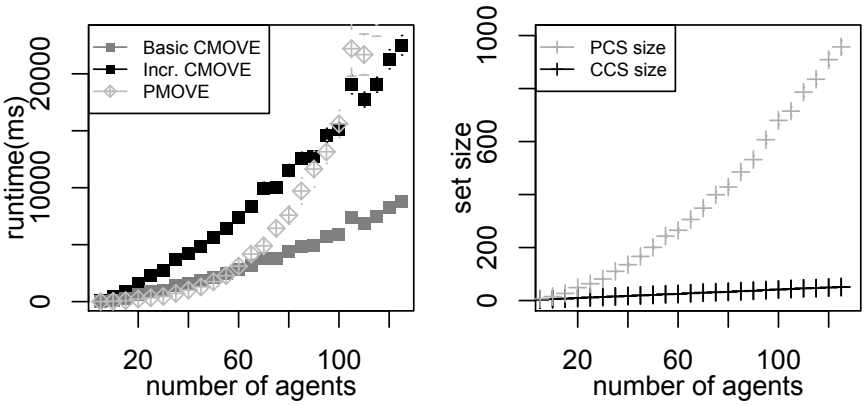


Figure 4.7: Runtimes (ms) for basic and incremental CMOVE, and PMOVE, in log-scale with the standard deviation of mean (error bars) (left) and the corresponding number of vectors in the PCS and CCS (right), for increasing numbers of agents.

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 4.7 (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 4.7 (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.

From these — as well as the Random Graphs results — we conclude that CMOVE

scales much better than PMOVE. Furthermore, we conclude that Basic CMOVE is faster than incremental CMOVE (at least for 8 objectives or less).

4.3.3 Convex AND/OR Tree Search

In Section 4.1.2, we described how the optimal joint action in a CoG can be found using an AND/OR Search tree (AOST), such as in Figure 4.3b, using AND/OR tree search (TS). In such an AOST, there are two types of nodes: OR nodes that represent agents, and AND nodes that represent actions. The TS algorithm traverses the tree, taking maximizations over the values of the subtrees rooted by the AND node children of an OR node, and summations over the values of subtrees rooted by the OR node children of an AND node. In this subsection, we show how the inner loop approach can be applied to create multi-objective methods from this algorithm.

First, we note that AND/OR tree search algorithms have been extended to compute a PCS by Marinescu (2009). When we apply this inner loop approach to compute the PCS to TS, it yields an algorithm we call *Pareto TS (PTS)*.[8] To define PTS, we must update Definition 26 to be a set of Pareto-optimal payoffs. We refer to such a subtree value set as an *intermediate PCS (IPCS)*.

Definition 31. *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 25) plus the (pruned) cross-sum of the intermediate PCSs of the subtrees rooted by the (OR-node) children of a_i:

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

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.

As we argued for CMOVE, computing a CCS rather than a PCS is often faster and sufficient for many real-world problems. To compute the CCS in a memory-efficient way, we propose *convex TS (CTS)*. CTS simply replaces `PPrune` by `CPrune` in Definition 31. 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

[8] Marinescu (2009) applies this approach to different algorithms from the class of AND/OR tree search algorithms as base algorithms. Here, we use TS, as it is the most memory-efficient.

subproblem in a sequential search scheme, regardless of whether that scheme is VE or TS; this is both more runtime and memory efficient.

Like in CMOVE, we can apply *incremental pruning* (Cassandra et al., 1997), i.e., pruning after every cross-sum. Incremental pruning is more memory efficient, as it greatly reduces the sizes of the cross-sums.

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.

Analysis

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

Corollary 4. *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 by Definition 2. 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.[9] For OR-nodes this is $O(|\mathcal{A}_{max}||ICCS_{max}|)$.

Corollary 5. *The space complexity of CTS is*

$$O(n|ICCS_{max}| + |\mathcal{A}_{max}||ICCS_{max}| + |ICCS_{max}|^c),$$

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. After computing a cross-sum at most $|ICCS_{max}|^c$ vectors are generated before pruning. □

Note that when incremental pruning is applied, the full cross-sums (of size $|ICCS_{max}|^c$) are never generated. Instead, the cross-sums of only 2 (instead of c) ICCSs are computed. CTS is much more memory efficient than CMOVE, which has a space complexity that is exponential in the induced width (Theorem 20).

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

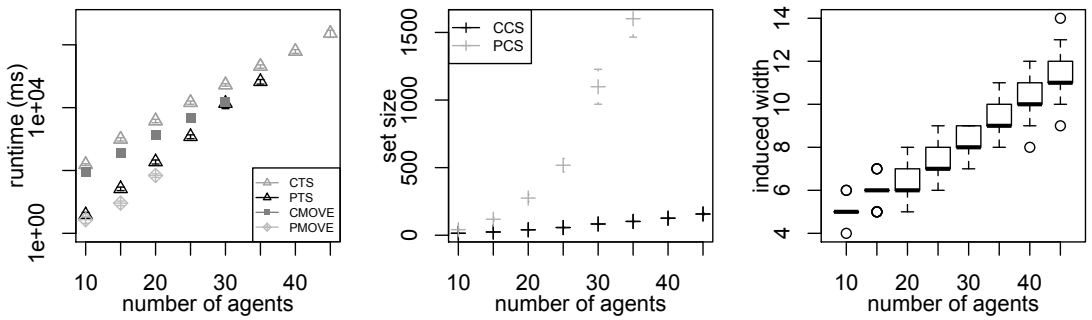


Figure 4.8: (left) Runtimes in ms of CTS, basic CMOVE and PMOVE on random 3-objective MO-CoGs with varying numbers of agents n and $\rho = 2.0n$ local payoff factors. (right) The corresponding CCS and PCS sizes of the MO-CoGs. (right) The corresponding induced widths of the MO-CoGs.

4.3.4 Experiments: CTS versus CMOVE

In this subsection we empirically compare CMOVE and CTS in settings where there is little available memory. In order to test the memory-efficiency, we limit the memory the algorithms is allowed to use, using the `-Xmx` runtime option for the JAVA Virtual Machine. All experiments use our JAVA implementation (version 1.6 SE), and the corresponding MacOS X Virtual Machine. All experiments in this subsection were run on a 2.4 GHz Intel Core i5 computer, with 4 GB memory.

To obtain the PTs for CTS, we use the same heuristic as CMOVE and VELS to generate an elimination order. We then transform this elimination order into a PT for which $m \leq w \log n$ holds (whose existence is guaranteed by Theorem 15), using the procedure suggested by Bayardo and Miranker .

We employ the same generation procedure for random graphs as in Section 4.3.2. 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 very challenging random graphs with a varying number of agents n, three objectives, and $\rho = 2.0n$ local payoff functions. We limited the maximal available memory to 10MB.

Figure 4.8 (left) shows that within 10 MB, PMOVE and CMOVE can solve MO-CoGs of respectively 20 and 30 agents. Because the local coverage sets while running CMOVE are smaller than those while running PMOVE, CMOVE requires less memory and therefore can tackle problems with higher numbers of agents. In this figure, we show only basic CMOVE and not incremental CMOVE because the runtimes of basic CMOVE are better, and incremental CMOVE was not able to solve larger problem instances.

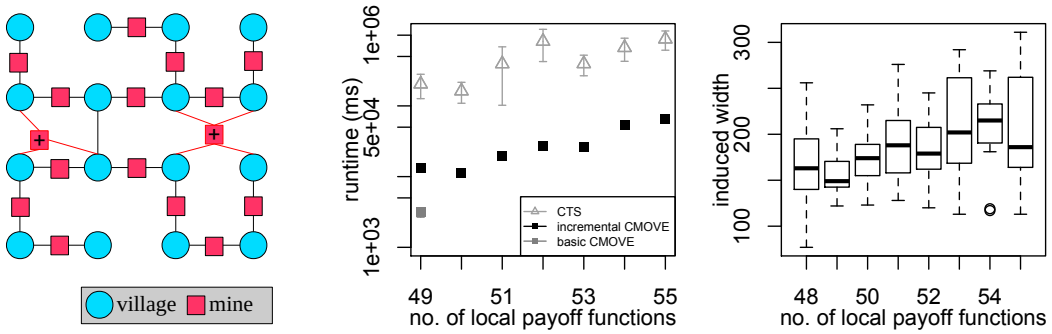


Figure 4.9: An example of a 4 by 4 Mining Field instance. The additional mines m are marked with a '+'.

Solving MO-CoGs with 35 agents or more, the induced width (shown in the right of Figure 4.8) becomes to large, and a memory-efficient approach is required. CTS and PTS are more memory efficient than CMOVE and PMOVE. However, as Figure 4.8 (middle) shows, the CCSs are much smaller than the PCSs. For example, at 35 agents, the CCSs consist of only 102 payoff vectors on average, while the PCSs consist of 1602 payoff vectors on average. Therefore, while PTS runs out of memory at 40 agents, CTS can solve problems with larger numbers of agents. In fact, we were unable to generate Random Graphs that made CTS run out of memory. We therefore conclude that CTS can tackle larger problems than PTS.

When comparing runtimes, we observe that CTS requires more runtime; 3.0 times more than CMOVE at 35 agents. However, CTS can handle more agents within the memory constraints. We therefore conclude that while CMOVE is faster than CTS, CTS offers a solution for MO-CoGs where CMOVE runs out of memory.

Mining Field

We compare the performance of basic and incremental CMOVE and CTS on a variation of Mining Day that we call *Mining Field*. We use Mining Field in order to ensure an interesting problem for the memory-restricted setting. In Mining Day (see Section 4.3.1), 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 CMOVE 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 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 4.9 (left). 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 basic and incremental CMOVE and CTS on Mining Field, we tested them on a 7×7 instance (49 agents) with 3 objectives, and 10MB available memory. 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 and even PTS, which ran out of memory for all problems.

Figure 4.9 (middle) shows that within 10 MB, basic CMOVE can solve MO-CoGs of only 49 local payoff functions or less. Contrary to our Random Graph results however, incremental CMOVE can solve larger problem instances than basic CMOVE. An explanation for this is, is that in the Mining Field experiment, the induced width (Figure 4.9 (right)) increases much more slowly than in the Random Graph experiment of Figure 4.8. We thus expect that the part that for the Mining Field experiment, the dominant factor in the memory requirements of CMOVE (Theorem 20) is the computation of the cross-sums of Equation 4.6, whilst in Random Graphs the dominant part is the storage of the new VSFs. This also explains why after 55 local payoff functions, both CTS — whose main memory requirement is the computation of the cross-sums — *and* incremental CMOVE run out of memory. We therefore conclude that incremental CMOVE is more memory-efficient than basic CMOVE, and that for problems with a low induced width CTS and incremental CMOVE use a comparable amount of memory.

Summarizing, we have shown that CMOVE and CTS offer different trade-offs between memory and runtime. Basic CMOVE is faster than Incremental CMOVE which is in turn faster than CTS. However, Basic CMOVE also uses most memory. Incremental CMOVE and CTS use less memory, and can therefore tackle larger problems. CMOVE (both basic and incremental) scales poorly in the induced width in terms of memory usage. CTS can solve MO-CoGs with a much higher induced width. However, both the memory requirements of CTS and of CMOVE depend on the size of the cross-sums between local/intermediate CCSs. In the following Section, we propose methods that scale much better in the size of these local/intermediate CCSs.

4.4 OLS for MO-CoGs

In the previous section, we showed how to create CCS methods by taking an inner loop approach and using a single-objective search schema such as VE or TS. Furthermore, we have shown that when ample memory is available CMOVE is faster, and therefore better, while if memory is limited, CTS can solve MO-CoGs that CMOVE cannot. In this section, we create novel CCS methods using an outer loop rather than an inner loop approach. Not only are outer loop methods easier to create than inner loop methods — as the single-objective search algorithms themselves do not have to change — and can take *any* single-objective CoG solver as a subroutine, but they also have better computational and space complexity bounds for small and medium numbers of objectives.

4.4.1 Variable Elimination Linear Support

Rather than dealing with the multiple objectives in the inner loop of the VE search schema, as CMOVE does, we can also employ our *outer loop* approach, i.e., the optimistic linear support we defined in Section 3.3, and employ VE as a subroutine. We refer to the resulting algorithm as *variable elimination linear support (VELS)*.

Because VELS uses the OLS outer loop, VELS builds the CCS incrementally, i.e., with each iteration of its outer loop, it 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 4.1.1) 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, $\mathbf{a}$, that is optimal for this CoG and its payoff vector, $\mathbf{u}(\mathbf{a})$ are then added to the partial CCS. Having dealt with the multiple objectives in the outer loop of OLS, VELS thus relies on VE to exploit the graphical structure in the inner loop.

VELS is implemented as $\texttt{OLS}(m, \texttt{SolveSODP}, \varepsilon)$ (Algorithm 5), where,

- m is a MO-CoG, and
- `SolveSODP` consists of two steps: first calling VE to retrieve the optimal action, $\mathbf{a}$ for a scalarized instance of m, and then evaluating this action to retrieve the corresponding payoff vector $\mathbf{u}(\mathbf{a})$.

The computational and space complexities can immediately be derived from combining the computational and space complexities of OLS (Theorems 4 and 6) with those of VE (Theorems 11 and 12).

Corollary 6. *The computational complexity of VELS*

$$O(\ (|\varepsilon\text{–}CCS| + |\mathcal{W}_{\varepsilon-CCS}|)(n|\mathcal{A}_{max}|^w + d\rho + R_{nw} + R_{heur})\ ),$$

where $|\mathcal{A}_{max}|$ is the maximal number of actions for a single agent *and w is the induced width, ρ is the number of factors, and R_{nw} and R_{heur} are the time it costs to run* `newCornerWeights`*, and* `maxValueLP`*, as defined in Section 3.3.*

Proof. This follows directly from filling in the runtime of VE, $n|\mathcal{A}_{max}|^w$, for R_{so}, and the look-up and summation of ρ local payoff vectors of length d for R_{pe}, in Theorem 4. □

The overhead of OLS itself, i.e., computing new corner weights, R_{nw}, and calculating the maximal relative improvement, R_{heur}, is small compared to the VE calls. $R_{nw} + R_{heur}$ and $d\rho$ are thus negligible in practice. When comparing the runtime of CMOVE and VELS, we observe that in the former, the runtime of VE is multiplied by the runtime of the pruning operators (R_1 and R_2 in Theorem 19), while in the latter, the runtime of VE is multiplied by $(|\varepsilon\text{–}CCS| + |\mathcal{W}_{\varepsilon-CCS}|)$. Considering that $|\mathcal{W}_{\varepsilon-CCS}|$ is linear in $|\varepsilon\text{–}CCS|$ for $d = 2$ and $d = 3$, while `CPrune` is polynomial in the size of the local CCSs, we thus expect VELS to be much faster for these numbers of objectives. However, because $|\mathcal{W}_{\varepsilon-CCS}|$ grows exponentially with the number of objectives (Theorem 5) we expect CMOVE to be faster than VELS for larger number of objectives.

Corollary 7. *The space complexity of VELS is*

$$O(d|\varepsilon\text{–}CCS| + d|\mathcal{W}_{\varepsilon\text{–}CCS}| + n\ |\mathcal{A}_{max}|^w).$$

Proof. This follows directly from filling in the memory requirements of VE, $n|\mathcal{A}_{max}|^w$, for M_{so}, in Theorem 4. The memory requirements for computing the payoff vector, $\mathbf{u}(\mathbf{a})$, given $\mathbf{a}$ are negligible. □

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 20).

4.4.2 Experiments: VELS versus CMOVE

We now empirically evaluate VELS in comparison to CMOVE. We no longer compare against the non-graphical method and PMOVE as these are clearly dominated by CMOVE. 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 subsection were run on a 2.4 GHz Intel Core i5 computer, with 4 GB memory.

Random Graphs

To test VELS on randomly generated MO-CoGs, we use the same MO-CoG generation procedure as in Section 4.3.2. To determine how the scalability of exact and approximate VELS compares to that of 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 4.10 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.

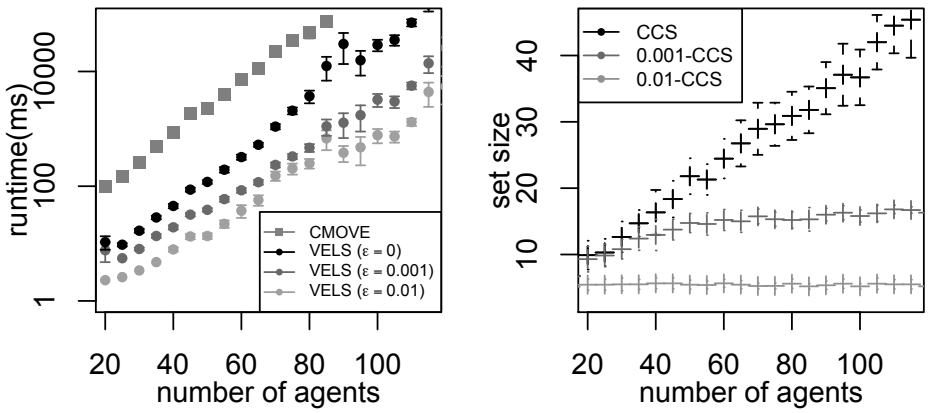


Figure 4.10: (left) The runtimes of CMOVE and VELS with different values of ε, for varying numbers of agents, n, and $\rho = 1.5n$ factors, 2 actions per agent, and 2 objectives and (right) the corresponding sizes of the ε-CCSs.

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, as we see in Figure 4.10 (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 average 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

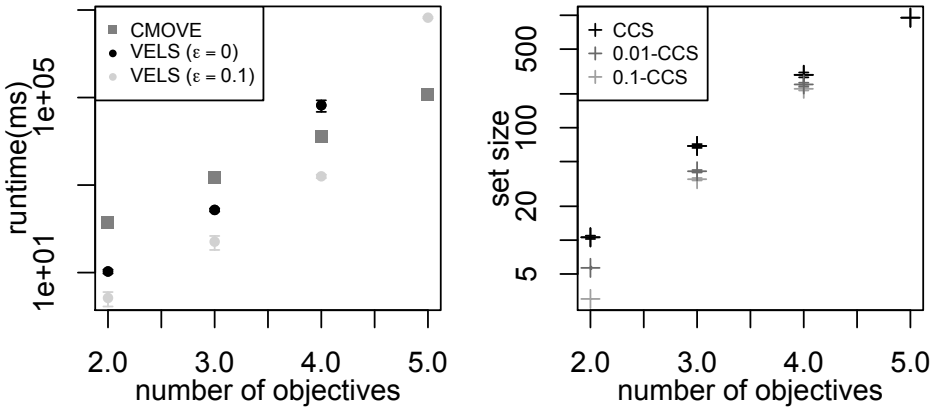


Figure 4.11: (left) the runtimes of CMOVE and VELS ($\varepsilon = 0$ and $\varepsilon = 0.1$), for varying numbers of objectives (right) the size of the ε-CCS for varying numbers of objectives.

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 4.11 (left) plots the number of objectives against the runtime (in log scale). Because the CCS grows exponentially with the number of objectives, as can be seen in Figure 4.11 (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 4.11 (right)) does not stabilize when the number of objectives grows, as can be seen in the following table:

$\lvert\varepsilon$–$CCS\rvert$	$\varepsilon = 0$	$\varepsilon = 0.001$	$\varepsilon = 0.01$	$\varepsilon = 0.1$
$d = 2$	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.[10] We note that these conclusions are conform expectations, because when we compare the runtimes of CMOVE and VELS

[10]Note that all the runtime figures in this subsection are in logscale.

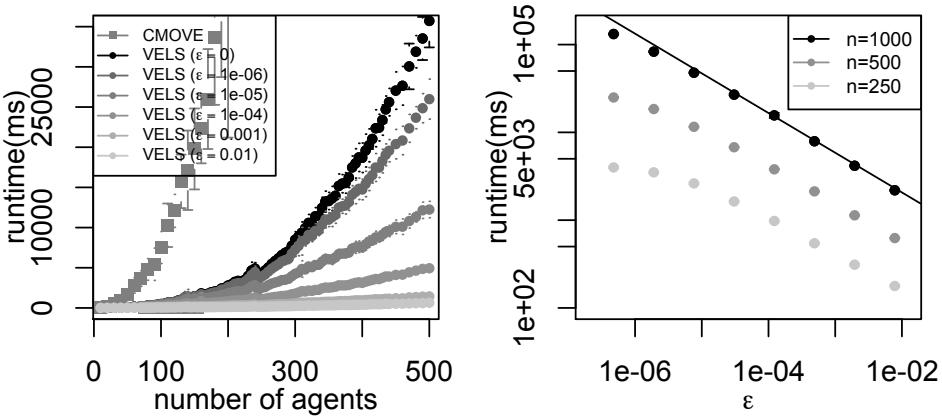


Figure 4.12: (left) plot of the runtimes of CMOVE and VELS with different values of ε, for varying n (up to 500). (right) loglogplot of the runtime of VELS on 250, 500, and 1000 agent mining day instances, for varying values of ε.

(Theorem 19 and Corollary 6), CMOVE is *polynomial* in the size of the (local) CCSs, where VELS is merely *linear* in the size of the CCS and the number of corner weights (which is in turn linear in the size of the CCS for $d = 2$ and $d = 3$).

Mining Day

We now compare CMOVE and VELS on the Mining Day benchmark using the same generation procedure as in Section 4.3.2. We generated 30 Mining Day instances for increasing n and averaged the runtimes (Figure 4.12 (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 4.12 (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.

4.4.3 AND/OR Tree Search Linear Support

VELS is considerably more memory efficient than CMOVE. However, CTS is more memory efficient than either of them. In order to become even more memory efficient than CTS — as well as faster for small numbers of objectives we define *tree search linear support (TSLS)*.

TSLS employs OLS with TS as the single-objective solver subroutine. TSLS is equal to 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 16) and the priority queue. Consequently, TSLS is even more memory efficient than CTS, while being faster for small and medium numbers of objectives.

Corollary 8. *The computational complexity of TSLS is*

$$O((|\varepsilon{-}CCS| + |\mathcal{W}_{\varepsilon-CCS}|)\ (n\ |\mathcal{A}_{max}|^m + R_{nw} + R_{heur})),$$

where $m \leq w \log n$ *and* $\varepsilon \geq 0$.

Proof. The proof is the same as that of Theorem 6 but with the time complexity of VE replaced by that of TS (Theorem 14). □

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 16) and the priority queue. VELS (Theorem 6) thus has much better space complexity than CMOVE (Theorem 20). 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.

Corollary 9. *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 Corollary 7 but with the space complexity of VE replaced by that of TS (Theorem 13). □

As mentioned in Section 4.1.2, 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 4 and 5 (CTS), and 8 and 9 (TSLS). The advantages of the outer loop methods compared to their corresponding inner loop methods however remain the same as for TSLS and CTS. Therefore, in this dissertation we focus on comparing the most memory-efficient inner loop method against the most memory-efficient outer loop method.

4.4.4 Experiments: TSLS versus VELS and CTS

In this section, we compare TSLS to CTS, and VELS and CMOVE[11]. In order to test the memory-efficiency, we limit the memory the algorithms is allowed to use, using the `-Xmx` runtime option for the JAVA Virtual Machine. All experiments use our JAVA implementation (version 1.6 (SE)), and the corresponding MacOS X Virtual Machine. As before, we use both random graphs and the Mining Field benchmark.

To obtain the PTs for CTS and TSLS, we again 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, using the procedure suggested by Bayardo and Miranker .

Random Graphs

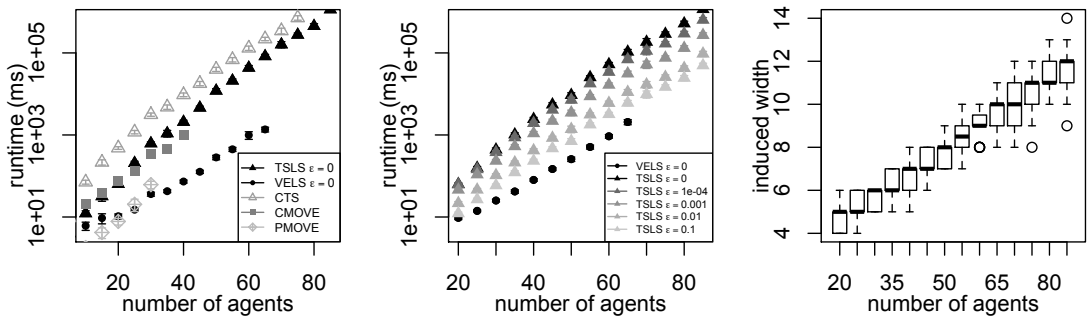
First, we test our algorithms on random graphs, employing the same generation procedure as in Section 4.3.2. Because connections between agents in these graphs are generated randomly, the induced width varies between different problems. On average, the induced width increases with the number of local 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 4.13a 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 4.13c) 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.

However, scaling beyond 65 agents requires a memory-efficient approach. Figure 4.13a 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.

[11] As in Section 4.4.2, where we write CMOVE, we mean basic CMOVE.



(a) (b) (c)

Figure 4.13: (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, 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 4.13b). 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.

Mining Field

We compare the performance of VELS against TSLS on *Mining Field* (as defined in Section 4.3.4). We no longer consider CTS and CMOVE because these inner loop methods have consistently higher runtime than their corresponding outer loop methods, and worse space complexity. We use Mining Field (as described in Section 4.3.4) in order to ensure an interesting problem for the memory-restricted setting.

To compare the runtimes and memory requirements, 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.

Figure 4.14 shows the comparison between VELS and TSLS. VELS runs out of memory only at 16 additional factors, at an induced width of 6. Compared to the random-graph results in Section 4.4.4, the induced widths of the problems that 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.

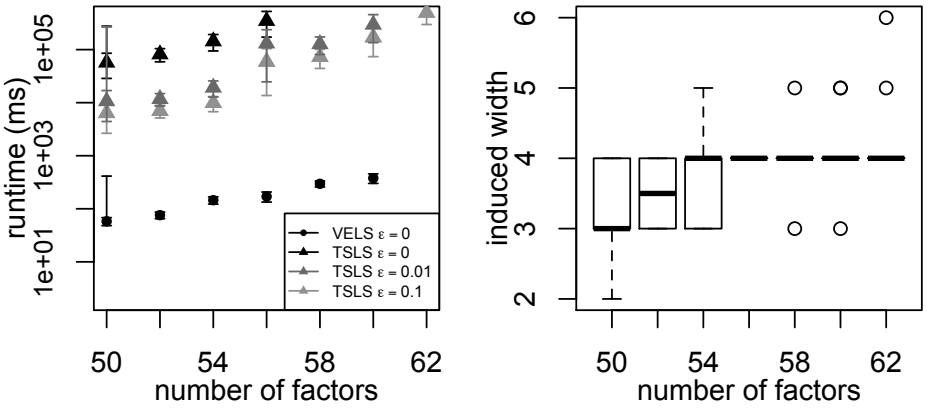


Figure 4.14: (left) Runtimes in ms of TSLS (for varying amounts of allowed error ε), VELS ($\varepsilon = 0$), and CMOVE on 2-objective Mining Field instances with varying numbers of additional mines $m \in [2..14]$ and a grid size of $s = 7$. (right) The corresponding induced widths of the Mining Field instances.

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 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.

4.4.5 Variational Optimistic Linear Support

VELS and TSLS are both exact outer loop methods, that offer different trade-offs with respect to speed and memory usage. Furthermore, it is also possible to sacrifice accuracy for runtime by allowing some slack ε to limit the number of calls to VE or TS. However, because VE and TS are exact single-objective solvers, this does not change the fact that VELS and TSLS have an exponential runtime in the induced width. To address this, we can instead also use an approximate single-objective solver as a subroutine.

In this subsection we propose using variational CoG solvers, as described in Section 4.1.3, as a basis for finding the CCS in a MO-CoG. As previously described, variational methods rely on restructuring and reparameterization rather than local summation and maximization (like VE and TS). Therefore, it is not obvious how to apply an inner loop

approach. However, as an outer loop approach does not affect the inner workings of the single-objective solvers, it is possible to use variational methods inside OLS. This is an important advantage for OLS, as variational methods are amongst the state-of-the-art solvers for CoGs.

Variational solvers for CoGs are bounded approximate algorithms. Therefore, we must use the OLS schema that allows for approximate subroutines, as described in Section 3.5. Using Algorithm 7, with a bounded-approximate variational method yields a new algorithm we call *variational optimistic linear support (VOLS)*. VOLS has two important advantages over VELS and TSLS. Firstly, because variational methods scale much better than VE and TS, VOLS achieves unprecedented scalability. In addition, since the variational method we use, which is called *weighted mini-buckets (WMB)* (Liu and Ihler, 2011), computes bounded approximations, VOLS does so too. Secondly, we leverage the key insight that VOLS can hot-start each call to WMB by reusing the *reparameterizations* output by WMB on earlier calls, as described in an abstract manner in Section 3.6.

In the rest of this subsection we describe how to implement reuse in OLS using variational methods in the VOLS algorithm. Specifically, we show how to use the reparameterizations output by a variational method as well as the lower bound for reuse in the next call to the variational solver. In Section 4.4.6 we compare VOLS empirically to CMOVE and VELS, and show that it scales much better than these methods at very little cost to the quality of the found solutions.

VOLS uses a variational subroutine as its instantiation of `approxSolveSODP`, to solve scalarized instances of the MO-CoG. This subroutine takes a scalarized MO-CoG as input. As output, the subroutine produces a lower-bound joint action $\mathbf{a}_l$, which we use to construct the approximate CCS. It also produces an upper bound $\bar{u}$ on the optimal value, which we use to bound the quality of the final approximate CCS, and to prioritize instances in the series of single-objective problems to solve. Furthermore, the variational method manipulates the set of scalarized local payoff functions $\mathcal{U}_{\mathbf{w}}$ to output a reparameterization, i.e., a set of manipulated local payoff functions $\mathcal{U}'_{\mathbf{w}}$ for which all joint actions have the same (scalar) payoff:

$$\forall \mathbf{a} \sum_{u^e \in \mathcal{U}_{\mathbf{w}}} u^e(\mathbf{a}_e) = \sum_{u^g \in \mathcal{U}'_{\mathbf{w}}} u^g(\mathbf{a}_g). \tag{4.8}$$

We can re-use the reparameterization, i.e., $\mathcal{U}'_{\mathbf{w}}$, to hot-start the reparameterization of a new scalarized instance for a new weight vector $\mathbf{z}$. Specifically, if we define the *difference graph* between two scalarization weights $\mathbf{w}$ and $\mathbf{z}$ as

$$\mathcal{U}_{\mathbf{w}\to\mathbf{z}} = \{u^e_{\mathbf{w}\to\mathbf{z}}(\mathbf{a}_e) = (\mathbf{z}-\mathbf{w}) \cdot \mathbf{u}^e(\mathbf{a}_e) : \mathbf{u}^e(\mathbf{a}_e) \in \mathcal{U}\},$$

then adding this difference graph to the reparameterization $\mathcal{U}'_{\mathbf{w}}$ yields a valid reparameterization for $\mathbf{z}$:

$$\hat{\mathcal{U}_{\mathbf{z}}} = \mathcal{U}'_{\mathbf{w}} \cup \mathcal{U}_{\mathbf{w}\to\mathbf{z}}.$$

When $\mathbf{w}$ is close to $\mathbf{z}$, the magnitude of the local payoff functions in $\mathcal{U}_{\mathbf{w}\to\mathbf{z}}$ is small, and $\hat{\mathcal{U}}_{\mathbf{z}}$ is close to $\mathcal{U}'_{\mathbf{w}}$. Intuitively, $\hat{\mathcal{U}}_{\mathbf{z}}$ is therefore likely to be closer to the eventual reparameterization $\mathcal{U}'_{\mathbf{z}}$ that the variational subroutine will produce for $\mathcal{U}_{\mathbf{z}}$, than $\mathcal{U}_{\mathbf{z}}$ itself would be, and fewer iterations of the variational method will be required to further tighten the bounds and find $\mathcal{U}'_{\mathbf{z}}$. Using the difference graph $\mathcal{U}_{\mathbf{w}\to\mathbf{z}}$ in order to hot-start the next call to a variational subroutine in OLS works for any variational single-objective CoG solver.

Using the definition of the difference graph, we can define variational optimistic linear support (VOLS) formally. VOLS — defined in Algorithm 12 — takes a MO-CoG $\langle \mathcal{D}, \mathcal{A}, \mathcal{U} \rangle$ and any variational single-objective coordination graph subroutine as input. In this section we refer to the variational subroutine as `variationalSOSolver`.

Following the OLS framework, VOLS keeps a set $\mathcal{S}$, that becomes an approximate CCS (line 1), a set of upper bounds on the optimal values that VOLS finds for scalarized instances (for individual $\mathbf{w}$), U_{old} (line 2), and starts looking for solutions, i.e., approximately optimal joint actions and payoffs, for the extrema of the weight simplex (line 3–4).

In order to enable the reuse of reparameterizations and lower bounds found by the variational subroutine, VOLS keeps a set $\mathcal{R}$ (line 5) with tuples of weights $\mathbf{w}$, and reparameterizations and joint actions that implement the lower bounds produced at those $\mathbf{w}$ by `variationalSOSolver` in iterations of the main loop.

In the main loop (lines 6–17), VOLS iteratively pops a corner weight $\mathbf{w}$ off the priority queue Q and solves the corresponding scalarized MO-CoG, $\mathcal{U}_{\mathbf{w}}$, as in standard OLS. However, instead of just calling the single-objective solver for $\mathcal{U}_{\mathbf{w}}$ directly, VOLS first looks for the reparameterization $\mathcal{U}'_{\mathbf{v}}$ found in earlier iterations (on line 8), for the weight closest to $\mathbf{w}$. Because, the value for all joint actions remains the same when after reparameterizing $\mathcal{U}_{\mathbf{v}}$ to $\mathcal{U}'_{\mathbf{v}}$ (Equation 4.8), adding the difference graph, $\mathcal{U}_{\mathbf{v}\to\mathbf{w}}$ results in a graph, $\hat{\mathcal{U}}_{\mathbf{w}} = \mathcal{U}'_v \cup \mathcal{U}_{\mathbf{v}\to\mathbf{w}}$, for which

$$\forall \mathbf{a} \sum_{u^e \in \mathcal{U}_{\mathbf{w}}} u^e(\mathbf{a}_e) = \sum_{u^h \in \hat{\mathcal{U}}_{\mathbf{w}}} u^h(\mathbf{a}_h).$$

In other words, reusing the reparameterization for $\mathbf{v}$ on the scalarized graph for $\mathbf{w}$ does not affect the scalarized payoff, $u_{\mathbf{w}}(\mathbf{a})$, for any $\mathbf{a}$, and $\hat{\mathcal{U}}_{\mathbf{w}}$ is thus a valid reparameterization $\mathcal{U}_{\mathbf{w}}$. The aim of applying this reparameterization is for $\hat{\mathcal{U}}_{\mathbf{w}}$ to be closer to the eventual output graph $\mathcal{U}'_{\mathbf{w}}$ of the variational solver. Our key insight is that by doing so, the runtime of the variational single-objective solver can be drastically reduced if the reparameterization $\mathcal{U}'_{\mathbf{v}}$ was found at a weight $\mathbf{v}$ close to $\mathbf{w}$. We verify this experimentally in Section 4.4.6.

Besides the reparameterization $\mathcal{U}'_v \cup \mathcal{U}_{\mathbf{v}\to\mathbf{w}}$, `variationalSolver` is also provided with the joint action $\mathbf{a}_{\mathbf{v}}$ that achieves the lower bound of the previous weight $\mathbf{v}$. This joint action can be reused as an initial *guess* for the joint action at $\mathbf{w}$. If at any time during the execution of `variationalSolver` for $\mathcal{U}'_v \cup \mathcal{U}_{\mathbf{v}\to\mathbf{w}}$, the upper bound is achieved by $\mathbf{a}_{\mathbf{v}}$, the variational solver can stop. Such lower bound reuse is thus highly effective when the variational single-objective solver can produce optimal solutions for the

Algorithm 12: VOLS($\langle \mathcal{D}, \mathcal{A}, \mathcal{U} \rangle$, variationalSOSolver)

1 $\mathcal{S} \leftarrow \emptyset$; // approximate CCS of multi-objective payoff vectors $\mathbf{u}(\mathbf{a})$
2 $U_{old} \leftarrow \emptyset$; // set of previous $\mathbf{w}$ and $\bar{u}_{\mathbf{w}}$, for determining optimistic estimates for new corner weights
3 $Q \leftarrow$ an empty priority queue ; // a priority queue with corner weights to search
4 Add extrema of the weight simplex to Q with infinite priority;
5 $\mathcal{R} \leftarrow \emptyset$; // set of reparameterizations, joint actions, and associated weights
6 **while** $\neg Q$.isEmpty() $\wedge \neg$*timeOut* **do**
7 $\quad \mathbf{w} \leftarrow Q$.dequeue(); // retrieve a weight vector
8 $\quad \mathcal{U}'_{\mathbf{v}}, \mathbf{a}_{\mathbf{v}} \leftarrow$ select previous reparameterization and joint action found for the closest weight $\mathbf{v}$ to $\mathbf{w}$ from $\mathcal{R}$;
9 $\quad \mathcal{U}'_{\mathbf{w}}, \ \mathbf{a}_l, \ \bar{u}_{\mathbf{w}} \leftarrow$ variationalSOSolver$(\mathcal{U}'_{\mathbf{v}} \cup \mathcal{U}_{\mathbf{v} \to \mathbf{w}})$; // a variational single objective solver.
10 $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{(\mathbf{w}, \mathcal{U}'_{\mathbf{w}}, \mathbf{a}_l)\}$; // store the reparameterization of the scalarized graph for reuse
11 $\quad U_{old} \leftarrow U_{old} \cup \{(\mathbf{w}, \bar{u}_{\mathbf{w}})\}$; // store upper bound for $\mathbf{w}$, for determining the next max. possible improv.
12 $\quad$ **if** $\mathbf{u}(\mathbf{a}_l) \notin \mathcal{S}$ **then**
13 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{u}(\mathbf{a}_l)\}$; // add lower bound payoff and associated action, $\mathbf{u}(\mathbf{a}_l)$, to the approximate CCS
14 $\quad\quad W \leftarrow$ compute new corner weights and max. possible improvements $(\mathbf{w}, \Delta_{\mathbf{w}})$ using U_{old} and $\mathcal{S}$;
15 $\quad\quad Q$.addAll(W);
16 $\quad$ **end**
17 **end**
18 **return** $\mathcal{S}$;

scalarized problem as it can circumvent the *decoding* phase of variational algorithms, which is often very computationally intensive.

The single-objective variational solver (called on line 9) produces three outputs: the new reparameterized graph $\mathcal{U}'_{\mathbf{w}}$, an upper bound on the optimal scalarized payoff, $\bar{u}_{\mathbf{w}}$, and the approximately optimal joint action $\mathbf{a}_l$. Note that $\mathbf{a}_l$ implies a lower bound on the optimal payoff in $\mathbf{w}$, i.e., $\mathbf{w} \cdot \mathbf{u}(\mathbf{a}_l)$. All of these are stored (lines 10 and 11).

If $\mathbf{u}(\mathbf{a}_l)$ is not already in $\mathcal{S}$, then it is added to it and new corner weights are identified. VOLS calculates the maximum possible improvement for the new corner weights by solving a linear program (line 14) based on the corner weights for which the variational subroutine has been called, and the upper bounds found at those weights, as described in Section 3.6. Finally — as is standard in OLS — the new corner weights are added to the priority queue Q (line 15), with the maximal possible improvements as priority. Because the maximum possible improvement to the scalarized payoff is guaranteed to be at one of the corner weights of $\mathcal{S}$ even when the single-objective subroutine is approximate (Theorem 7), VOLS terminates when Q is empty.

Upon termination, we can use U_{old} to determine the approximation quality ε, of the approximate CCS, $\mathcal{S}$ using the following corollary of Corollary 2:

Corollary 10. *VOLS returns $\mathcal{S}$, an ε-CCS, where*

$$\varepsilon = \max_{(\mathbf{w},\bar{u}_{\mathbf{w}})\in U_{old}} \big(\bar{u}_{\mathbf{w}} - u^*_{\mathcal{S}}(\mathbf{w}) \big).$$

Variational solvers for single-objective CoGs typically have no runtime guarantees, but are anytime and provide bounded approximations. Because OLS inherits the runtime guarantees of the single-objective solvers it employs, VOLS has no runtime guarantees either. In our implementation we stop the single-objective solver after a limited number of iterations. However, when the lower bound becomes equal to the upper bound during an iteration of the variational single-objective solver, it can terminate before this maximum is reached.

Because VOLS employs reuse of the reparameterizations, the memory requirements are are proportional to the number of weights for which a reparameterization is stored. This storage happens on line 10, in the main loop of OLS. This happens for every weight for which the variational single-objective CoG is called, i.e., $(|\varepsilon{-}CCS| + |\mathcal{W}_{\varepsilon-CCS}|)$ times. The reparameterizations we use in our implementation are of (roughly) the same size as the original graphs. However, different variational solvers can have different reparameterization sizes.

When the memory burden of reuse becomes too big there are two possible ways to get around it: either not to do reuse, and use the variational method inside standard OLS (Algorithm 5), or to store the reparameterizations on disk rather than in memory.

To summarize, VOLS is a bounded approximate method that employs variational CoG solvers inside OLS with reuse (Algorithm 8). These variational CoG solvers have been proven to be very effective, but have no optimality or runtime guarantees. They are however anytime, and can be stopped after a number of iterations. Because VOLS does not have runtime guarantees, we have to test its effectivity empirically. Furthermore, though reuse can easily be motivated intuitively — as we have done in this subsection — it does not come with any guarantees that it is in fact faster to do so. The effectiveness of reuse must therefore be evaluated empirically as well.

4.4.6 Experiments: VOLS versus VELS

In this subsection, we compare the performance of VOLS that of VELS, as well as a version of VOLS without reuse. We no longer compare against CTS, CMOVE, and TSLS, as these methods are slower than VELS. All experiments in this subsection use our JAVA implementation (version 1.6 (SE)). This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative, on a virtual machine on the HPC Cloud service, with an 2.7 GHz Intel processor. We use random graphs as well as the Mining Day benchmark.

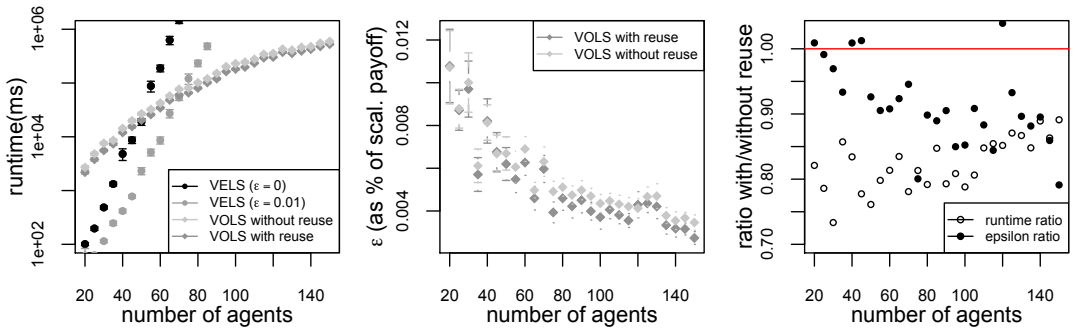


Figure 4.15: (left) The runtime (in logscale) of VOLS versus the runtime (in logscale) of VELS as a function of the number of agents n with $\rho = 1.8n$ and $d = 3$. The error bars represent SDOM. (middle) The quality (ε) of the approximate CCSs produced by VOLS with and without reuse for the same MO-CoGs. (right) The ratio of the runtimes and ε of VOLS with and without reuse for the same MO-CoGs.

Random Graphs

In this section, we compare the performance of VELS and VOLS on randomly generated MO-CoGs. For the single-objective subroutine, we use *weighted mini-buckets (WMB)* (Ihler et al., 2012; Liu and Ihler, 2011), with an i-bound of $i = 1$. $i = 1$ is the highest degree of approximation.[12] We limit the maximal number of iterations of the single-objective subroutine at 100. The MO-CoGs are generated following the procedure explained in Subsection 4.3.2.

We compare VELS and VOLS on random 3-objective MO-CoGs with increasing numbers of agents n with $\rho = 1.8n$ factors per agent. We use more objectives and a higher ρ than we used in the previous (sub)sections, in order to create more challenging problems, i.e., problems with a more rapidly increasing induced width and with larger CCSs. We generated 25 MO-CoGs for each number of agents and ran both algorithms on the same instances.

Figure 4.15 (left) shows that exact VELS ($\varepsilon = 0$) is faster than VOLS for the random graphs with up to 55 agents. However, the runtime of VELS (for any ε) increases exponentially with the induced width, whilst that of VOLS does not. At 70 agents, VOLS is more than an order of magnitude faster than exact VELS, and at 150 agents VOLS is still faster than exact VELS is at 70 agents.

Of course, VOLS only produces an ε-CCS, whereas exact VELS produces an exact one. However, when we measure ε using Corollary 10, we find that it is consistently 1.1% of the value or less. Note that this ε needs to be determined after VOLS finishes, as VOLS does not guarantee it can reach a given ε, but instead is just limited at a maximum number of iterations of its single-objective subroutine. When we introduce comparable ε into VELS, by allowing a slack of ε of 1% (i.e., $\varepsilon = 0.01$), we see that VELS remains faster than VOLS up until 75 agents. However, the runtime of

[12]The code was provided to us by dr. Alexander Ihler.

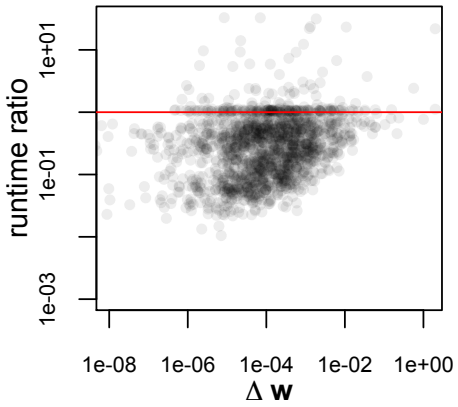


Figure 4.16: The runtime of the variational subroutine for different weights with reuse, divided by the runtime without reuse in logscale, as a function of the difference with the closest weight $\Delta\mathbf{w}$, for a MO-CoG with $n = 125$ with $\rho = 1.8$ and $d = 3$.

VELS still increases exponentially with the induced width. Furthermore, VELS uses exponential memory, while VOLS does not. In Figure 4.15 (left), we see that VELS with $\varepsilon = 0.01$ can solve problems up until 85 agents, but then runs out of memory. This is a significant memory limitation, as we used a virtual machine with 4GB memory (the default setting for JAVA). In comparison, VOLS uses very little memory —comparable to the size of the input graph — and does not run out of memory. Therefore, we conclude that VOLS scales much better than VELS, even when VELS is allowed a comparable ε to the ε-bounds that VOLS guarantees in practice.

When we zoom in on the ε-bound that VOLS guarantees for MO-CoGs with rising numbers of agents in Figure 4.15 (middle), ε appears to decrease as a function of the size of the problem. At 150 agents, VOLS (with reuse) produced an ε-CCS with a ε of only 0.27% of the scalarized payoff. We thus conclude that VOLS' improved scalability in practice comes at only a negligible cost in terms of payoff, even for larger graphs.

To test the effect of reuse on runtime, we compare the runtime of VOLS with and without reuse. We ran both versions on the same 25 instances for each number of agents. Figure 4.15 (left) shows that VOLS with reuse requires consistently less runtime across all numbers of agents. Across all numbers of agents, VOLS with reuse is a factor 1.22 faster. Furthermore, Figure 4.15 (middle) shows that VOLS with reuse produces ε-CCSs with a consistently smaller ε. Figure 4.15 (right), shows the ratio of the runtimes of VOLS with, and without reuse (the runtime with reuse divided by the runtime without reuse), and the ratio of the ε produced by VOLS with and without reuse. While the runtime ratio gradually increases, meaning less benefit from reuse, the ε ratio gradually decreases, meaning better accuracy. Furthermore, VOLS with reuse has lower runtime and ε overall. Therefore, we conclude that for random graphs, reuse contributes positively to VOLS' performance.

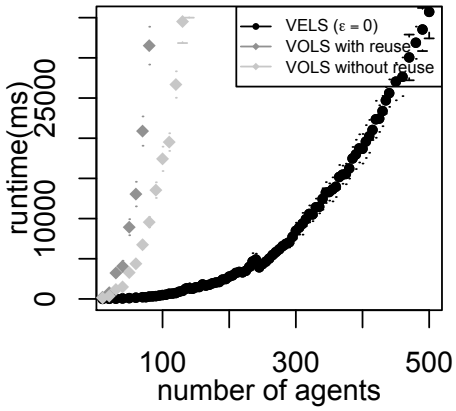


Figure 4.17: Plot of the runtimes of VOLS (with and without reuse) and VELS ($\varepsilon = 0$) with different values of ε, for varying n.

In order to test our hypothesis that reuse is more effective when the reparameterization reused for a weight $\mathbf{w}$ was found at a weight $\mathbf{z}$ closer to $\mathbf{w}$, we tested the effect of reuse on the runtime of the single-objective subroutine inside VOLS. For this purpose, we used a single MO-CoG with $d = 3$, $n = 125$ and $\rho = 1.8n$. For each weight $\mathbf{w}$ in the sequence, we executed the variational subroutine with and without reuse. The average total runtime with reuse was $0.10s$ while it was $0.16s$ without reuse. Figure 4.16 shows the ratio between the runtime with and without reuse as a function of $\Delta\mathbf{w} = |\mathbf{z} - \mathbf{w}|$, i.e., the Euclidean distance between the current weight on the weight on which the reused reparameterization is based. This figure shows that the runtime is positively correlated with $\Delta\mathbf{w}$. However, there are also a lot of weights for which reuse has little or no effect, and even outliers for which reuse has a negative effect on the runtime. These outliers contribute disproportionally to the average runtime: although they make up only 5% the weights, they are responsible for 48% of the total runtime of VOLS with reuse. For comparison, the first 5% of the calls, i.e., those with the 5% largest $\Delta\mathbf{w}$, account for only 9% of the runtime. We therefore conclude that our hypothesis seems correct on average, but that a small $\Delta\mathbf{w}$ is not a guarantee for a low runtime.

Mining Day

In order to compare VOLS and VELS on a highly structured problem, with limited induced width, we use the Mining Day benchmark. We use the same generation procedure as in Section 4.3.2, with 2-5 workers per village and a connectivity of 2-4 mines per village. We generated 25 Mining Day instances for increasing n and averaged the runtimes.

Because the induced width of this problem is limited by the maximum number of mines per village, the induced width does not increase with the number of agent.

Furthermore, because we use a maximum connectivity of 4, the induced width is also low.

In Figure 4.17, we see that exact VELS ($\varepsilon = 0$) outperforms VOLS — both with and without reuse. At 220 — at which point we stopped running VOLS — VOLS was 147 (with reuse) respectively 42 (without reuse) times slower than VELS. We therefore conclude that in problems with a low induced width, VOLS does not scale better than VELS in the number of agents.

When we inspect how VOLS with reuse compared to VOLS without reuse, we see that VOLS without reuse seems to perform better in terms of runtime than VOLS with reuse. However, the ε of VOLS with reuse is better than without reuse: the maximal ε for VOLS with reuse was 1.5×10^{-6} and has an average of 1.5×10^{-9}, against a maximum of 1.0×10^{-3} and an average of 8.7×10^{-6} for VOLS. We therefore conclude that for Mining Day, reuse makes VOLS slower rather than faster, although it slightly improves the ε.

4.5 Conclusion

In this chapter, we proposed several novel algorithms for computing a CCS for MO-CoGs. These new methods either follow an inner loop approach (CMOVE and CTS), or an outer loop approach based on OLS (VELS, TSLS, and VOLS).

When we compare our inner loop methods to corresponding inner loop methods that compute the PCS, we find that our CCS methods scale much better in the size of the MO-CoG as well as in the number of objectives. Therefore, we conclude that computing a CCS should be preferred over computing a PCS whenever possible. As we have argued in Chapter 2, the CCS applies to both the case when the scalarization function is linear and to the case when stochastic policies are allowed.

When we compare inner loop methods to corresponding OLS-based methods, i.e., methods that use the same single-objective search schema as a basis, we find that inner loop methods scale better in the number of objectives, while OLS-based methods scale better in the number of agents. We argue that the latter is usually more important in practice.

When compared our exact methods, i.e., CMOVE, CTS, VELS and TSLS. CMOVE and VELS are fastest. For 2 and 3 objectives, VELS is faster than CMOVE. VELS also scales better in the number of agents. However, CMOVE scales better in the number of objectives. Both CMOVE and VELS use an exponential (in the induced width) amount of runtime, while CTS and TSLS use an amount of memory independent of the induced width. Between CTS and TSLS, TSLS is most memory efficient.

When exact CCSs are not required, VELS and TSLS (but not CMOVE and CTS) can produce ε-CCSs for any pre-specified ε. Doing so can result in a decrease in runtime of orders of magnitude, even for small ε.

When it is not necessary to pre-specify a given ε, VOLS outperforms the other methods in terms of runtime and memory use. As we have seen from our random

graph experiments, VELS and TSLS use an amount of runtime and memory that is exponential in the induced width, while VOLS does not. However, when the induced width is limited (as in Mining Day) VELS is typically faster than VOLS, even when VELS solves the MO-CoG exactly ($\varepsilon = 0$).

To summarize, we have proposed five different CCS algorithms for MO-CoGs. These methods are typically much faster than corresponding PCS methods. Compared to each other, our methods provide different trade-offs between runtime, memory, and output quality (ε).

Chapter 5

Sequential Decision-Making

In the previous chapter, we have considered problems in which a single (joint) action is performed in order to obtain a reward. However, many decision problems consist of a sequence of decisions. This sequence of decisions typically takes place in an environment that is affected by these decisions. Therefore, the agents do not only have to consider their immediate reward, but also the reward they will be able obtain later, by changing the state of the environment to a more favorable one.

In this dissertation, we consider two multi-objective sequential single-agent decision-theoretic models. The first model is fully observable and extends the single-objective *Markov decision process (MDP)* (Bellman, 1957b; Puterman, 1994; Sutton and Barto, 1998; Wiering and Van Otterlo, 2012). The second is partially observable and extends the *partially observable Markov decision process (POMDP)* (Cassandra, 1998; Kaelbling et al., 1998; Spaan, 2012). The single-objective version of these models are widely used and applied in areas such as: communication networks (Altman, 2002), planning and scheduling (Scharpff et al., 2013), games (Szita, 2012) and robotics (Kober and Peters, 2012). The multi-objective models have been gaining traction relatively recently. There is a significant body of work on *multi-objective Markov decision problems (MOMDPs)* (Roijers et al., 2013a), and recently, there have been several papers on *multi-objective partially observable Markov decision problems (MOPOMDPs)* (Soh and Demiris, 2011a,b; Roijers et al., 2015c; Wray and Zilberstein, 2015).

In this chapter we propose new methods for MOMDPs and MOPOMDPs based on OLS. We focus on challenging instances of these models, and show that state-of-the-art methods for single-objective planning can be effectively employed as subroutines in OLS in order to create multi-objective methods, with relatively little effort. This is a major advantage, for it implies that if the state-of-the-art for single-objective methods improves, then — via OLS — so does the state-of-the-art in multi-objective methods.

For MOMDPs, we focus on finite-horizon problems with a very large state-space. Specifically, we employ the multi-objective version of the *maintenance planning problem (MPP)* (Scharpff et al., 2013; Roijers et al., 2014a; Scharpff et al., 2015, 2016)

as an illustrative example problem. The main challenge in both the single- and multi-objective MPP is the size of the state and action spaces. For this problem, a highly problem-specific solution method exists which makes use of a SPUDD (Hoey et al., 1999) encoding. An inner loop method that uses such a problem-specific encoding (and SPUDD itself) as a base for a multi-objective method, takes a lot of time to create. Furthermore, the construction of the inner loop method has to be redone for every different problem that requires such a problem-specific solution method or when a state-of-the-art method for the single-objective version of a problem is invented. In Section 5.2, we show that we can create effective multi-objective methods based on OLS that do not have these disadvantages. We illustrate this by exchanging SPUDD and its encoding by the CoRe algorithm which we proposed recently (Scharpff et al., 2016). Furthermore, we show that by using an approximate solver, i.e., UCT* instead of an exact solver, ε-CCSs can be found in a fraction of the runtime it takes to compute an exact CCS.

For POMDPs, we focus on infinite-horizon problems. Creating planning methods for single-objective POMDPs is an active field of research (Kurniawati et al., 2008; Pineau et al., 2006; Poupart et al., 2011), and as such, is regarded a hard and open problem. Therefore, extending the POMDP model to the multi-objective setting, i.e., *multi-objective POMDPs (MOPOMDPs)*, results in a very computationally challenging problem. Maybe this is one of the reasons why only a few papers about MOPOMDPs exist. In Section 5.3, we propose the first planning method for multi-objective POMDPs that computes a CCS and scales reasonably well. In order to do so, we use point-based planning methods (Shani et al., 2013), and specifically Perseus (Spaan and Vlassis, 2011), as a single-objective subroutine. We show that it is highly beneficial to employ *reuse* (Section 3.6) of the value functions found in previous OLS iterations. In order to enable this reuse — and to avoid separate policy evaluations, which can be expensive in POMDPs — we show how to create OLS-compliant versions of point-based planning methods, that use a multi-objective representation of the value function.

The rest of this chapter is structured as follows. We introduce MDPs, MOMDPs, POMDPs and MOPOMDPs in Section 5.1. In Section 5.2, we show how we can create efficient methods for large MOMDPs, using the MPP as an illustrating problem. In Section 5.3 we propose *OLS with Alpha Reuse (OLSAR)*, which is our main contribution for MOPOMDPs. We conclude in Section 5.4, with a summary of our results.

5.1 Background

In this section we provide the requisite background on the MOMDP and MOPOMDP models. However, we first describe their single-objective counterparts.

5.1.1 Markov decision processes

First, we describe the fully observable setting.

Definition 32. *A (single-objective)* Markov decision process (MDP) *(Wiering and Van Otterlo, 2012) is a tuple* $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ *where,*

- $\mathcal{S}$ *is the state space, i.e., the set of possible states the environment can be in,*
- $\mathcal{A}$ *is the action space, i.e., the set of actions the agent can take,*
- $T : S \times A \times S \to [0, 1]$ *is the transition function, giving the probability of a next state given an action and a current state,*
- $R : S \times A \times S \to \mathbb{R}$ *is the reward function, specifying the expected immediate expected scalar reward corresponding to a transition.*

At each timestep t the agent observes the current state of the environment $s \in \mathcal{S}$. When the agent takes an action $a \in \mathcal{A}$ the environment transitions to a new state s'.

The state in an MDP is *Markovian*, i.e., the current state s of the environment and the current action of the agent a are a sufficient statistic for predicting the next transition probabilities $T(s'|s, a)$ and the associated expected immediate reward. The agent's history, i.e., the states and actions that led to the current state, do not provide additional information in that respect.

The agent's goal in an MDP is to find a *policy* that maximizes the expected sum of future rewards. Informally, a policy is a set of decision rules that for each point in time and state of the MDP, prescribes how to choose an action to perform. The expected sum of future rewards given a policy, π, is called the value of π, V^π. In a *finite-horizon setting* there is a limited number of timesteps, h, and the sum is typically undiscounted:

$$V^\pi = E[\sum_{t=0}^{h} R(s_t, a_t, s_{t+1})|\pi, \mu_0],$$

where μ_0 is the distribution over initial states s_0. In a *discounted infinite-horizon setting*, the number of timesteps is not limited, but there is a discount factor, $0 \leq \gamma \leq 1$, that specifies the relative importance of future rewards with respect to immediate rewards:

$$V^\pi = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})|\pi, \mu_0].$$

There are different choices for what to condition a policy on. We refer to the set of all possible policies for an MDP as Π. A stationary policy is one that conditions the actions only on the current state, $\mathcal{S} \times \mathcal{A} \to [0, 1]$, i.e., for each state there is a probability distribution over actions. We refer to the set of all possible stationary policies as Π^S.

Non-stationary policies can besides the state also condition on the timestep t.[1] Another special type of policy is a *deterministic* policy. A deterministic policy only assigns 0 or 1 probabilities to actions for each value of what it conditions on. A deterministic stationary policy thus specifies a single action in each state $\mathcal{S} \to \mathcal{A}$. We refer to the set of all possible deterministic policies as Π^D, and the set of all deterministic stationary policies as Π^{DS}.

For infinite horizon MDPs we can restrict the search to deterministic stationary policies in order to find an optimal policy. For finite horizon MDPs we can restrict restrict the search to deterministic non-stationary policies.

Theorem 21. *(Boutilier et al., 1999; Howard, 1960) For any infinite-horizon single-objective MDP, there exists a deterministic stationary policy, $\pi^* \in \Pi^{DS}$ that is optimal.*

Theorem 22. *(Boutilier et al., 1999) For any finite-horizon single-objective MDP, there exists a deterministic policy $\pi^* \in \Pi^D$ that is optimal.*

The existence of an optimal deterministic policy drastically reduces the size of the space of policies that need to be considered. In the case of finite-horizon MDPs there are $|\Pi^D| = |A|^{h|S|}$ policies to consider, and for infinite-horizon MDPs even $|\Pi^{DS}| = |A|^{|S|}$, while there would be infinitely many if we consider stochastic policies as well. Optimal policies in MDPs can be found efficiently using e.g., dynamic programming techniques (Bellman, 1957a) or linear programming (Manne, 1960).

Definition 33. *A* multi-objective Markov decision process (MOMDP) *(Roijers et al., 2013a) is a tuple $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where,*

- *$\mathcal{S}$, $\mathcal{A}$, and T are the same as in an MDP,*
- *$\mathbf{R} : S \times A \times S \to \mathbb{R}^d$ is the reward function, specifying the expected immediate* vector-valued *reward corresponding to a transition.*

When only deterministic policies are allowed and the scalarization function, f, is nonlinear, non-stationary policies can be better than the best stationary ones. Therefore, unlike in single-objective MDPs, it is not always the case that we only need to consider deterministic stationary policies.

Theorem 23. *(White, 1982) In infinite-horizon MOMDPs, deterministic non-stationary policies can Pareto-dominate deterministic stationary policies that are undominated by other deterministic stationary policies.*

To see why, consider the following infinite-horizon MOMDP with discount factor γ, which was adapted from an example by White (1982): there is only one state, s and three actions a_1, a_2, and a_3, which yield rewards $(3, 0)$, $(0, 3)$, and $(1, 1)$, respectively. The transitions (for all actions) from state s all lead back to the same state. If we

[1] Note that there are also other options to condition on, e.g., the entire history of states and actions. For sake of brevity we not consider these here.

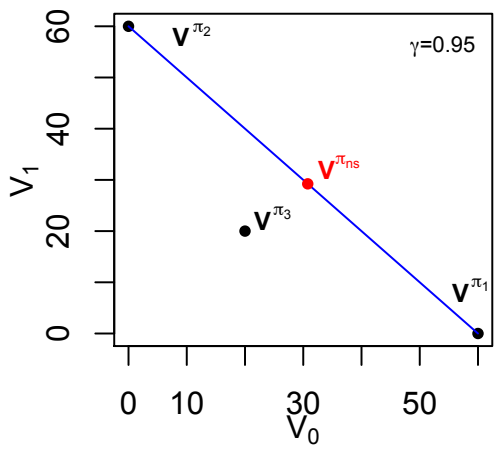


Figure 5.1: The policy values of π_1, π_2, π_3, and π_{ns} for the example MOMDP by White (1982), for $\gamma = 0.95$. The blue line indicates the possible values for all possible mixture policies that use π_1 and π_2 as base policies.

allow only deterministic stationary policies, then there are only three possible policies π_1, π_2, π_3, each corresponding to always taking one of the actions, all of which are Pareto optimal. These policies have the following values: $\mathbf{V}^{\pi_1} = (3/(1-\gamma), 0)$, $\mathbf{V}^{\pi_2} = (0, 3/(1-\gamma))$, and $\mathbf{V}^{\pi_3} = (1/(1-\gamma), 1/(1-\gamma))$. However, if we now consider the set of possibly non-stationary policies, we can construct a policy π_{ns} that alternates between a_1 and a_2, starting with a_1, whose value is $\mathbf{V}^{\pi_{ns}} = (3/(1-\gamma^2), 3\gamma/(1-\gamma^2))$. Consequently, this policy Pareto-dominates π_3, $\pi_{ns} \succ_P \pi_3$ when $\gamma \geq 0.5$ (Figure 5.1) and thus we cannot restrict our attention to stationary policies. Note that similar examples can be constructed for the finite-horizon setting.

For linear f however, we can restrict our attention to only deterministic stationary policies (Roijers et al., 2013a):

Theorem 24. *For an infinite-horizon MOMDP m, any $CCS(\Pi^{DS})$ of deterministic stationary policies is also a $CCS(\Pi)$ for stochastic non-stationary policies.*

Proof. This proof is similar to that of Theorem 1 and Theorem 3. If f is linear, we can translate the MOMDP to a single-objective MDP, for each possible $\mathbf{w}$. This is done by treating the inner product of the reward vector and $\mathbf{w}$ as the new rewards, and leaving the rest of the problem as is. Since the inner product distributes over addition, the scalarized returns remain additive. Thus, for every $\mathbf{w}$ there exists a translation to a single-objective MDP, for which an optimal deterministic and stationary policy must exist, due to Theorem 21. Consequently, when the optimal deterministic stationary policy for $\mathbf{w}$ is $\pi_{\mathbf{w}}$, there cannot exist a non-stationary and/or stochastic policy, π', such that $\mathbf{w} \cdot \mathbf{V}^{\pi'} > \mathbf{w} \cdot \mathbf{V}^{\pi_{\mathbf{w}}}$. Because this holds for every possible $\mathbf{w}$, a CCS with respect to all deterministic stationary policies is also a CCS for all stochastic non-stationary policies. □

Any deterministic stationary CCS is thus sufficient for solving infinite-horizon MOMDPs with linear f, even when stochastic and non-stationary policies are allowed. For finite-horizon MOMDPs a similar theorem holds but for deterministic (possibly non-stationary) policies only:

Theorem 25. *For a finite-horizon MOMDP m, any $CCS(\Pi^D)$ of deterministic policies is also a $CCS(\Pi)$ for stochastic policies.*

Proof. The proof is identical to that of Theorem 24. □

In the proof of Theorem 24 we show than an MOMDP can be scalarized with a linear weight $\mathbf{w}$, resulting in an MDP. Because this fulfills the conditions of Assumption 1), we can use a deterministic CCS as a compact representation of a PCS for stochastic policies (Corollary 1). In Figure 5.1 for example, the values in the PCS of stochastic policies for the abovementioned MOMDP example by White (1982) are indicated by the blue line; these values can all be attained by mixing π_1 and π_2.

5.1.2 Partially Observable Markov Decision Problems

So far we have assumed that the state of the environment can be observed by the agent. However, in many planning problems this is not the case. For example, imagine a robot that has only a front camera as sensor input and needs to go to a given location in an office building. When driving around, many locations in such a building may look identical (especially in the hallways). Therefore the agent controlling the robot has to rely on the memory of its previous observations to disambiguate the state as much as possible. The *partially observable Markov decision process (POMDP)* (Kaelbling et al., 1998) is a decision-theoretic model that incorporates partial observability into a single-agent sequential decision problem.

Definition 34. *A POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O \rangle$ where,*

- *$\mathcal{S}$, $\mathcal{A}$, T, and R are the same as in an MDP,*
- *Ω, is the set of possible observations, and*
- *O is the observation function, giving the probability of each observation given an action and the resulting state, i.e., $O(o|a, s')$ is the probability of observing $o \in \Omega$ when taking action a resulting in state s' (regardless of the state in which action a was taken). It is thus a mapping: $\mathcal{A} \times \mathcal{S} \times \Omega \to [0, 1]$.*

Because the agent cannot observe the entire state, the history of previous observations yields information about the present state that is not contained in the latest observation. In other words, the observations the agent receives are not Markovian, and only conditioning on the last observation could lead to loss of optimality. This makes the planning problem in POMDPs much harder than in an MDP. In fact, infinite-horizon POMDPs are in general undecidable (Madani et al., 2003). However, there is some

leverage that can be used for planning. Specifically, it turns out that the *belief state* of an agent *is* a sufficient statistic for the (action-observation) history of an agent, and can be used as a Markovian state signal.

Definition 35. *The belief state $b(s)$, specifies for each state $s \in \mathcal{S}$ the probability of being in that state s.*

In a POMDP, an agent can maintain such a belief $b(s)$ (Åström, 1965). We call the belief of an agent at timestep 0, $b_0(s)$. The agent updates its belief when it takes an action a, makes an observation o, according to the following update rule:

$$b_{t+1}(s') = \frac{O(o|s',a)}{P(o|b_t,a)} \sum_{s \in \mathcal{S}} T(s'|s,a) b_t(s),$$

where,

$$P_b(o|b_t,a) = \sum_{s' \in \mathcal{S}} O(o|s',a) \sum_{s \in \mathcal{S}} T(s'|s,a) b_t(s).$$

We can use the belief updates to specify a transition function for belief-states, $T_b(b'|a,b)$. Specifically, in a given belief b_t, the next belief b_{t+1} only depends on the action a and the observation o, and the probability of an observation is $P(o|b_t,a)$. Furthermore, we can also specify a reward function $R_b(b,a) = \sum_{s \in \mathcal{S}} b(s) R(s,a)$, where $R(s,a) = \sum_{s' \in \mathcal{S}} R(s,a,s') T(s'|s,a)$. Together, the belief-state, R_b, and T_b can be used to specify a belief-MDP.

Theorem 26. *(Spaan, 2012) For each POMDP $\langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O \rangle$, there is an equivalent belief-MDP $\langle \mathcal{S}_b = \Delta\mathcal{S}, \mathcal{A}, T_b, R_b \rangle$, where $\Delta\mathcal{S}$ is the belief simplex, i.e., all valid probability distributions over $\mathcal{S}$, $\mathcal{A}$ is the same set of actions as in the POMDP, and T_b and R_b are as specified above. An optimal policy for this belief-MDP is an optimal policy for the POMDP, and the optimal value function for the belief-MDP is equal to the optimal value function for the POMDP.*

Corollary 11. *For any discounted infinite-horizon (single-objective) POMDP, there exists an optimal deterministic stationary policy conditioned on the belief. For any finite-horizon (single-objective) POMDP, there exists an optimal deterministic policy conditioned on the belief.*

Proof. This follows directly from Theorems 21, 22, and 26. □

The value function for a single-objective POMDP, $V(b)$, can thus be defined in terms of the belief state. The value function can be represented by a set $\mathcal{V}$ of α*-vectors*. Each vector α (of length $|S|$) gives a value for each state s. The value of a belief b given $\mathcal{V}$ is:

$$V(b) = \max_{\alpha \in \mathcal{V}} b \cdot \alpha. \tag{5.1}$$

Equation 5.1 implies the following observation:

Observation 3. *(Sondik, 1971) The value function of a POMDP is a* piecewise-linear and convex (PWLC) *function in* b.

Each α-vector is associated with an action. Therefore, a set of α-vectors $\mathcal{V}$ also provides a *policy* $\pi_{\mathcal{V}}$ that for each belief takes the maximizing action in Equation 5.1.

Because infinite-horizon POMDPs are in general undecidable, the exact optimal value function and accompanying α-vectors cannot be found exactly within finite time. An ε-approximate value function can in principle be computed using techniques like *value iteration (VI)* (Monahan, 1982) and *incremental pruning* (Cassandra et al., 1997). Unfortunately, these methods scale poorly in the number of states. However, *point-based* methods (Shani et al., 2013), which approximate the value function for the entire belief simplex by iteratively computing the best α-vector only for a set B of sampled beliefs, scale much better. We explain how this works in Section 5.3.1.

Definition 36. *An MOPOMDP is a tuple* $\langle \mathcal{S}, \mathcal{A}, T, \mathbf{R}, \Omega, O \rangle$ *where,*

- $\mathcal{S}$*,* $\mathcal{A}$*,* T*,* Ω*, and* O *are the same as in a POMDP,*
- $\mathbf{R} : S \times A \times S \to \mathbb{R}^d$ *is the reward function, specifying the expected immediate* vector-valued *reward corresponding to a transition.*

For MOPOMDPs, we observe that because of the equivalence between POMDPs and belief MDPs (Theorem 26 and Corollary 11), Theorem 24 also extends to POMDPs, i.e.,

Corollary 12. *For an MOPOMDP, any CCS of deterministic stationary policies — with respect to belief-state — is also a CCS for stochastic non-stationary policies.*

Furthermore, because MOPOMDPs can be scalarized to a POMDP equivalently to scalarizing a MOMDP, i.e., by taking the inner product of a weight vector $\mathbf{w}$ and the reward function $\mathbf{R}$ as the scalarized reward function, and leaving the rest of the problem intact, the CCS of deterministic policies can be used as a compact representation of the PCS of stochastic policies (Corollary 1).

The only current method available for computing a CCS for a MOPOMDP is based on an old method for MOMDPs. White and Kim (1980) proposed a method to translate a multi-objective MDP into a POMDP, which can also be applied to a MOPOMDP and yields a single-objective POMDP. Intuitively, this translation assumes there is only one "true" objective. Since the agent does not know which objective is the true one, this yields a single-objective POMDP whose state is a tuple $\langle s, i \rangle$ where s is the original MOPOMDP state and $i \in \{1 \dots d\}$ indicates the true objective. The resulting POMDP's state space is of size $|S|d$. The observations provide information about s but not about i. The resulting POMDP can be solved with standard methods but only those that do not require an initial belief, as such a belief would fix not only a distribution over s but also over i, yielding a policy optimal only for a given $\mathbf{w}$. Since this precludes the effective usage of solvers like point-based methods, it is a severe limitation.

For PCSs, no theorem like Corollary 12 exists. In fact, the only current research we are aware of that aims to find a PCS for a given MOPOMDP is by Soh and Demiris (2011a,b) who use evolutionary methods to approximate the PCS. However, evolutionary methods are heuristic algorithms that provide no guarantees regarding the quality of the approximation. Furthermore, as we discuss in Section 2.3, finding a PCS is often not necessary. In Section 5.3, we show how to employ the class of *point-based planning algorithms* (Shani et al., 2013) — to which many state-of-the-art planning methods for POMDPs belong — as subroutines in OLS, to create planning methods for MOPOMDPs and how these methods can provide bounds on the quality of the approximation.

5.2 OLS for Large Finite-Horizon MOMDPs

In this section, we show how to tackle challenging real-world MOMDPs using OLS. The problem we use to illustrate this is called the *maintenance planning problem (MPP)* (Scharpff et al., 2013; Roijers et al., 2014a; Scharpff et al., 2015, 2016). We first introduce the MPP and explain why it is a computationally hard problem. Then we briefly discuss the existing single-objective methods for the MPP, explain the intuition behind how these are able to solve the MPP efficiently, and how to use them as subroutines inside OLS. Finally, we compare the different subroutines in combination with OLS on instances of the MPP.

5.2.1 The Maintenance Planning Problem

The *maintenance planning problem* (MPP) (Scharpff et al., 2013; Roijers et al., 2014a; Scharpff et al., 2015, 2016) is a task-based planning problem in which a government needs to have road maintenance tasks performed. The government wants to minimize maintenance costs that it will ultimately have to pay (objective 1) while minimizing the hindrance of these maintenance tasks to traffic (objective 2). In order to do so, they place fines on causing traffic delay.

In order achieve their objectives, the local government uses a public tender to attract a group of contractors. Each contractor bids on a set of predetermined maintenance tasks. The tender is based on a dynamic-VCG mechanism (Bergemann and Valimaki, 2006; Cavallo, 2008). In this mechanism, the rewards for a contractor are a given weighted sum (i.e., a linear scalarization with a given weight $\mathbf{w}$) of the payment it receives from the government minus the costs of the maintenance tasks and the fines. Because of the mechanism, the optimal strategy for each contractor is to be truthful about the costs, and to cooperate in planning a policy for performing the maintenance tasks. In other words, because of the mechanism, the problem becomes a fully cooperative multi-agent MDP. Therefore, we will refer to the contractors as the *agents*, and to the multi-agent MDP as the single-objective MPP. A condition for being able to use the mechanism however, is that the optimal policy can be computed.

The government has estimates of the costs for each contractor based on previous tenders. However, in previous tenders, there was too much traffic delay. Therefore, the government wants to incentivize performing a policy that results in less hindrance for traffic by adding to the fines on traffic delay, i.e., they want to investigate the effect on the value of the optimal policies (i.e., the total costs and the total traffic delay) by changing $\mathbf{w}$. This can be modelled as finding the CCS a multi-objective multi-agent MDP. After finding the CCS, the local government can select a $\mathbf{w}$ for which the optimal policy of the contractors corresponds to the value vector the government selected from the CCS. We refer to this multi-objective multi-agent MDP as the multi-objective MPP.

It is important that the CCS can be computed exactly. Otherwise, the dynamic-VCG mechanism no longer guarantees that rational agents are truthful (Scharpff et al., 2013), and we can no longer model the MPP as a multi-objective MMDP. Therefore, we focus on using exact planning methods. However, approximate methods could still be used by the local government to get a global idea about the values in the CCS.

In the multi-objective MPP, each task has a minimal completion time. Depending on conditions that become known at the moment the task is started, a task may delay with a given amount of time. The probability of delay is known.

The hindrance for traffic depends on which tasks are performed concurrently. For instance, concurrent maintenance in a busy area causes more hindrance than when maintenance tasks are performed sequentially.

The maintenance tasks for a given agent i are a fixed predetermined set. For each task, a, there are

- a number of consecutive time steps required to complete the task, $d \in \mathbb{Z}^+$,
- the probability that a task will delay, $p \in [0, 1]$,
- the additional duration if the task is delayed, $\hat{d} \in \mathbb{Z}^+$, and
- a cost function, $c(a, t)$, that captures the cost of maintenance (e.g., personnel, materials, etc.) per time step $t \in [1, \ldots, h]$.

The reward for executing a maintenance task is a (time-independent) payment, minus these costs. This is the first objective in the multi-objective MPP. Note that the costs may differ for each time step (e.g., it may be more expensive to perform a task at night or during holiday periods). The total cost of the execution of a task is the sum of the costs for each time step the task is being executed including delays. Agents are restricted to executing a single task per time step and they can only plan activities if they are guaranteed to finish them within the plan horizon. Because the payments are independent of time, while the costs are dependent of time, the agents want to schedule their tasks at the cheapest time-slots. However, there is also an incentive for agents to cooperate and not schedule too many tasks in parallel, because of the fines that are incurred by traffic hindrance.

The hindrance caused by maintenance is given by the function $\ell(\mathcal{A}^t, t)$, expressed in traffic time lost (TTL). In this function, $\mathcal{A}^t$ is used to denote any combination of

maintenance tasks performed at time t, by one or more agents. This function is typically based on traffic simulations, but can also be computed using heuristic rules. TTL is the second objective in the multi-objective MPP.

In the multi-objective MPP, the state is defined as the activities yet to be performed (for all agents together), the time left to perform them, and the availability of the individual agents (as they may currently be executing a task). The possible actions for a single agent are starting a task that has not been performed yet (if not busy), continue a task (if busy), or do nothing (if not busy). For more details on the MPP please refer to (Scharpff et al., 2013).

The main challenge in the MPP is that both the size of the state space and the size of the action space are exponential in the number of agents (Boutilier, 1996). Furthermore, contrary to *decentralized* models (Becker et al., 2003; Nair et al., 2005; Witwicki and Durfee, 2010), the policy for each agent needs to condition on the entire (exponentially sized) state space in order to be optimal, as agents can predict the actions of other agents better based on the full state than based on only their own local state (as in decentralized models) (Scharpff et al., 2016). Therefore, planning needs to take place in a *centralized* manner, i.e., as if a single agent controls all others. It is, in general, not possible to tell how much value would be lost if a decentralized approach were taken as an approximation. Fortunately though, this does not mean that is impossible to exploit the structure of the MPP. It is however necessary to formulate tailored solution methods to the specific properties of the MPP.

5.2.2 Solving MPP instances

In the single-objective version of the MPP, a monetary value is assigned to each hour of traffic time lost (TTL), thereby linearly scalarizing the problem. For this single-objective problem, we discuss three solution methods as proposed by Scharpff et al. (2013), Roijers et al. (2014a) and (Scharpff et al., 2016). We later use these methods as the single-objective subroutine in OLS, in order to solve the multi-objective MPP. The single-objective methods are

- **SPUDD + compact encoding** — The *stochastic planning using decision diagrams (SPUDD)* algorithm (Hoey et al., 1999), is a value iteration algorithm that employs *algebraic decision diagrams (ADDs)* to represent value functions and policies. ADDs are compact representations that use state features — such as whether a task has already been performed or not in the MPP — to describe the value function. By doing so, many states are (implicitly) grouped together, leading to compact representations of the value function (or policy). SPUDD implements dynamic programming (Bellman, 1957a) via manipulation of these ADDs. Therefore, SPUDD can handle large state spaces well, as long as these state spaces are structured.

  However, the MPP has a large action space as well as a large state space. Therefore, Scharpff et al. (2013) propose an encoding of the single-objective MDP

that reduces the size of the action space (at the cost of slightly increasing the size of the state space). In this encoding, one time step is split up in one time step per agent. At each such a "sub time step" an agent commits to carrying out an action from its own available actions (i.e., tasks to perform, continue or do nothing), and augmenting this commitment to the state. In this manner, an MDP that SPUDD can handle is created.

SPUDD is an exact single-objective planning method, and can thus be used in conjunction with the dynamic-VCG mechanism.

- **CoRe** — We have recently proposed a new method (Scharpff et al., 2016) that exploits several of the properties of the single-objective MPP problem: firstly, we observe that if we separate the state space into local states per agent (i.e., which tasks an agent has left to perform, and whether it is currently busy), the transitions in these local states only depend on the actions of that local agent, i.e., there is *transition independence*. Secondly, we observe that the rewards that depend on more than one agent (i.e., the traffic delay increase/decrease due to concurrent execution of tasks) are sparse, as only tasks that are performed on roads that are relatively close affect each other, i.e., there is only sparse interaction between agents. Finally, we observe that because the tasks are only performed once, the interaction between agents can cease to exist when the particular tasks that interact have been performed.

  We proposed a new optimal solver for transition-independent multi-agent MDPs (such as the MPP), called CoRe (Scharpff et al., 2016). CoRe decomposes the returns into local subsets that it represents compactly in so-called *conditional return graphs (CRGs)*. Using CRGs the value of a joint policy and the bounds on partially specified joint policies can be efficiently computed. CoRe employs a branch-and-bound policy search schema building on CRGs.

  Like SPUDD, CoRe is an exact planning method, and can thus be used in conjunction with the dynamic-VCG mechanism. CoRe can solve much larger MPP instances than SPUDD. Therefore, it is vitally important to be able to use CoRe as a basis for a multi-objective method instead of SPUDD.

- **UCT* + compact encoding** — The UCT* algorithm (Keller and Helmert, 2013) combines the popular *upper-confidence bounds applied to trees (UCT)* planning algorithm (Kocsis and Szepesvári, 2006) with heuristic AND/OR search (AO*), in order to create a fast Monte Carlo tree search algorithm. We use this algorithm, together with the aforementioned encoding of the single-objective MDP by Scharpff et al. (2013), as an anytime approximate alternative to SPUDD, i.e., we stop it after a limited amount of runtime and use the best policy found so far.

  UCT* is an approximate planning method. Therefore, it cannot be used in practice on the MPP to make the dynamic-VCG mechanism work. However, it does

provide approximations that the government could use to get an idea about the range of the values in the CCS quickly.

It would be difficult to create a multi-objective MPP solver using an inner loop approach for all three of the above-mentioned single-objective solvers; it is not clear how ADDs and their manipulations could be extended to apply to sets of value vectors inside SPUDD, and in UCT* and CoRe it is unclear how the bounds that make these algorithms fast would function with vector-valued rewards.

When we employ the above-mentioned algorithms as subroutines inside OLS, we do not need to change the inner workings of these algorithms. Instead, we obtain an optimal policy (for SPUDD and CoRe) or an approximate policy (for UCT*), and run policy evaluation to obtain the corresponding value vector. In the next subsection, we compare the resulting multi-objective solution methods empirically.

5.2.3 Experiments: MPP

In order to demonstrate the efficacy of our algorithms, we test and compare OLS+SPUDD, OLS+UCT*, and OLS+CoRe on instances of the multi-objective MPP.[2] We use Joris Scharpff's MPP generator to create instances of this problem. We employ a JAVA implementation of OLS that calls the SPUDD package[3] for OLS+SPUDD or the PROST package[4] (Keller and Eyerich, 2012) for UCT*. These packages are freely available online. For CoRe we use our own JAVA implementation (as we did in Scharpff et al. (2016)).

Because we cannot let UCT* run until all states have been evaluated, we can only obtain partial policies. In our experiments we 'repair' partial policies, i.e., supplement these policies by taking a conservative action; in the MPP this means inserting "do nothing" actions.

All experiments in this subsection were conducted on DAS4 cluster[5] of Delft University of Technology, with a 2.4 GHz dual quad-core processor and 24 GB memory.

Comparing OLS-based methods

We use the single-objective subroutines of Section 5.2.2 in combination with OLS, and compare the resulting OLS+SPUDD and OLS+CoRe algorithms on instances of the MPP that both can solve within 30 minutes. These are 2- and 3-agent problems.[6] The results are shown in Table 5.1. Note that because SPUDD cannot solve many 3-agent problems, there are signicantly fewer instances for that number of agents.

[2] These are new experiments that were performed in collaboration with Joris Scharpff. The experiments will appear in his forthcoming dissertation as well.

[3] `https://cs.uwaterloo.ca/~jhoey/research/spudd/index.php`

[4] `http://prost.informatik.uni-freiburg.de/`

[5] `http://www.asci.tudelft.nl/pages/about-asci/das.php`

[6] Scharpff et al. (2016) show that CoRe can also handle 4-agent problems, but SPUDD cannot.

	$N = 2$			$N = 3$		
Algorithm	**Runtime (s)**	$\|CCS\|$	ε	**Runtime (s)**	$\|CCS\|$	ε
OLS+SPUDD	184.032	6.592	-	1997.079	7.591	-
OLS+CoRe	591.167	6.592	-	1160.516	7.591	-
OLS+UCT* (0.01s)	2.212	3.460	0.321	2.723	3.227	0.371
OLS+UCT* (0.1s)	3.432	4.615	0.209	4.572	3.561	0.263
OLS+UCT* (1s)	8.656	5.798	0.073	14.918	5.621	0.168
OLS+UCT* (10s)	39.906	6.690	0.002	85.191	7.364	0.022
OLS+UCT* (20s)	57.173	6.690	0.002	139.571	7.727	0.014

Table 5.1: Average runtimes, CCS sizes, and ε for 71 2-agent instances and 22 3-agent instances of the multi-objective MPP with 2 to 4 tasks per agent, and plan horizons between 3 and 10.

For 2-agent problems, which both algorithms can solve completely, OLS+SPUDD takes a factor 3.2 less time than OLS+CoRe. However, OLS+CoRe scales better in the number of number of agents, and is a factor 1.7 faster for the 3-agent instances. When we zoom in on the runtimes, as a function of the plan horizon in Figure 5.2, we observe that for 2 agents, the runtimes seem exponential in the plan horizon, and OLS+SPUDD is consistently faster than OLS+CoRe. For 3-agent problems however, OLS+CoRe is consistently faster than OLS+SPUDD. We therefore confirm that OLS+CoRe is better when there are more than 2 agents.

Finally, we compare the exact methods to the approximate OLS+UCT* method. Because UCT* does not provide ε-bounds, we cannot guarantee a bounded approximation. However, when we give each run of UCT* a fixed amount of runtime to solve a scalarized MPP, we can measure ε empirically by comparing the approximate CCSs outputted by OLS+UCT* to the true CCS outputted by OLS+SPUDD and OLS+CoRe. From Table 5.1, we observe that OLS+UCT* can produce results with little error in less than 10% of the runtime. E.g., for the 3-agent instances with 10 seconds per scalarized instance for UCT* achieves $\varepsilon = 2.2\%$ in $85s$ while CoRe needs $1160s$ on average to compute the exact CCS. Note that for some of the smaller instances, i.e., few agents and a small plan horizon, UCT* sometimes finds the optimal policy for the scalarized problems before its time-limit is reached and terminates before its time is up. This happened for 2 agents and $10s$ or $20s$ per call to UCT*. Therefore, we conclude that OLS+UCT* can be used to provide quick approximations if necessary. However, because we can only measure ε w.r.t. to the true CCS, we cannot know ε when we apply only OLS+UCT* (if it does not produce the exact CCS).

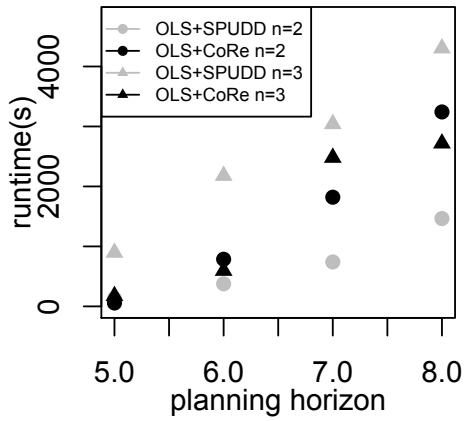


Figure 5.2: Average runtimes of OLS+SPUDD and OLS+CoRe on 2-agent instances and 3-agent instances of the multi-objective MPP with 2 to 4 tasks per agent, as a function of the plan horizon. NB: there are less data points for the larger planning horizons as this plot contains only those instances that both OLS+SPUDD and OLS+CoRe could solve.

5.3 OLS with Alpha Reuse for MOPOMDPs

In this Section we aim to create planning methods for *multi-objective partially observable Markov decision processes (MOPOMDPs)* (Soh and Demiris, 2011a,b; Roijers et al., 2015c; Wray and Zilberstein, 2015). Little research has been conducted on planning methods for MOPOMDPs; the naive approach to translate the MOPOMDP to a single-objective POMDP with an augmented state space (White and Kim, 1980), which we discussed in Section 5.1.2, precludes the use of POMDP methods that exploit an initial belief such as point-based methods (Shani et al., 2013), because specifying an initial belief fixes the scalarization weights. Another naive approach would be to use Random Sampling (RS) of scalarization weights in an outer loop approach (as discussed in Section 3.1.2). However, because POMDP planning is expensive, such an approach is undesirable, as it lacks guarantees with respect to approximation quality, and it might solve many scalarized instances that contribute little or nothing to the improvement of the output approximate CCS. Lastly, the evolutionary approach proposed by Soh and Demiris (2011a,b) does not have any guarantees with respect to approximation quality either, and computes the PCS of deterministic policies, and not the CCS, which we argue (see Section 2.3) is often a more desirable solution concept.

In this section, we propose a new MOPOMDP solution method called *optimistic linear support with alpha reuse (OLSAR)*. Our approach is based on *optimistic linear support (OLS)* (Chapter 3), and therefore selects its scalarization weights in such a way as to maximally reduce the maximal possible error of the intermediate CCS at each iteration, and — using a bounded approximate single-objective solver — produces an ε-CCS.

OLSAR contains two key components that — in addition to the usage of OLS — are essential to making it tractable for MOPOMDPs. First, it uses a novel *OLS-compliant* version of the point-based solver *Perseus* (Spaan and Vlassis, 2011). OLS-compliant Perseus solves a given scalarized POMDP and simultaneously computes the resulting policy's multi-objective value. Doing so avoids the need for the separate policy evaluation step that is often employed by OLS (e.g., as in Section 5.2). Since POMDP policy evaluation is computationally expensive, the use of OLS-compliant Perseus is key to OLSAR's efficiency. Second, rather than solving each scalarized POMDP from scratch, OLSAR reuses the α-matrices that represent each policy's multi-objective value to form an initial lower bound for subsequent calls to OLSAR-compliant Perseus. In this section we show how reuse — which we discussed in an abstract manner in Section 3.6 — can be implemented in the context of MOPOMDPs.

The rest of this section is structured as follows: first, we briefly discuss single-objective point-based planning methods in Section 5.3.1; Perseus (Spaan and Vlassis, 2011) in particular. Then, in Section 5.3.2 we introduce the OLSAR algorithm, as well as the OLS-compliant Perseus algorithm that OLSAR uses as a subroutine. In Section 5.3.3 we empirically test OLSAR, and show that the usage of OLS is key to keeping MOPOMDPs tractable and that reuse leads to dramatic reductions in runtime in practice.

5.3.1 Point-based POMDP methods

An infinite-horizon single-objective POMDP (Kaelbling et al., 1998; Madani et al., 2003) is a sequential decision problem that incorporates uncertainty about the state of the environment, and is specified as a tuple $\langle S, A, R, T, \Omega, O, \gamma \rangle$. As we discussed in Section 5.1.2, an agent typically maintains a belief b over which state it is in. The value function for a single-objective POMDP, V_b, is defined in terms of this belief and can be represented by a set $\mathcal{A}$ of *α-vectors*. Each vector α (of length $|S|$) gives a value for each state s. The value of a belief b given $\mathcal{A}$ is:

$$V_b = \max_{\alpha \in \mathcal{A}} b \cdot \alpha. \tag{5.2}$$

Each α-vector is associated — or *tagged*, as we did in MO-CoGs in Chapter 4 — with an action. Therefore, a set of α-vectors $\mathcal{A}$ also provides a *policy* $\pi_{\mathcal{A}}$ that for each belief takes the maximizing action in Equation 5.2.

While infinite-horizon POMDPs are in general undecidable (Madani et al., 2003), an ε-approximate value function can in principle be computed using techniques such as *value iteration (VI)* (Monahan, 1982) and *incremental pruning* (Cassandra et al., 1997). Unfortunately, these methods scale poorly in the number of states.

Point-based methods (Shani et al., 2013), which perform approximate backups by computing the best α-vector only for a set B of sampled beliefs, scale much better in the number of states than VI and incremental pruning. Point-based methods perform a series of so-called *point-based backups*. For each $b \in B$, a point-based backup is

performed by first computing for each a and o, the *back-projection* $g_i^{a,o}$ of each next-stage value vector $\alpha_i \in \mathcal{A}_k$:

$$g_i^{a,o}(s) = \sum_{s' \in S} O(a, s', o) T(s, a, s') \alpha_i(s'). \tag{5.3}$$

Because $g_i^{a,o}(s)$ are identical for all $b \in B$, these $g_i^{a,o}(s)$ can be cached and shared between different b. However, once new $\alpha_i(s)$ are computed, new $g_i^{a,o}(s)$ also need to be computed.

For each b, the back-projected vectors $g_i^{a,o}$ are used to construct $|A|$ new α-vectors (one for each action):

$$\alpha_{k+1}^{b,a} = r^a + \gamma \sum_{o \in \Omega} \arg\max_{g^{a,o}} b \cdot g^{a,o}. \tag{5.4}$$

Finally, the $\alpha_{k+1}^{b,a}$ that maximizes the inner product with b (in Equation 5.2) is retained as the new α-vector for b:

$$\texttt{backup}(\mathcal{A}_k, b) = \arg\max_{\alpha_{k+1}^{b,a}} b \cdot \alpha_{k+1}^{b,a}. \tag{5.5}$$

Point-based methods typically perform several point-based backups using $\mathcal{A}_k$ for different b to construct the set of α-vectors for the next iteration, $\mathcal{A}_{k+1}$ in order to enable efficient caching of the $g_i^{a,o}(s)$ vectors. By constructing the α-vectors only from the $g^{a,o}$ that are maximizing for the given b, point-based methods avoid generating the much larger set of α-vectors that an exhaustive backup of $\mathcal{A}_k$ would generate.

In this section, we use the popular Perseus (Spaan and Vlassis, 2011) point-based planning algorithm as a basis for creating a multi-objective method. Perseus is given in Algorithm 13. It takes as input an initial lower bound $\mathcal{A}$ on the value function in the form of a set of α-vectors, a set of sampled beliefs, a scalarization weight $\mathbf{w}$, and a precision parameter η. The initial lower bound vectors in $\mathcal{A}$ are typically determined heuristically. Perseus repeatedly improves upon this lower bound (lines 3-10) by finding an α-vector that improves upon the value, $V_b = b \cdot \alpha$, for each sampled belief. However, this does not imply that a point-based backup is performed for each b in the sampled belief set B. Instead, Perseus selects a random belief from the set of sampled beliefs (line 6) and, if possible, finds an improving α-vector for it (line 7). When such an α-vector also improves the value for another belief point in B, this belief point is ignored until the next iteration (line 9). This results in an algorithm that generates few α-vectors in early iterations, but improves the lower bound on the value function, and gets more precise, i.e., generates more α-vectors, in later iterations. This is an important aspect of the algorithm, as the number of α-vectors in $\mathcal{A}_k$ at the start of iteration k is the determining factor for the runtime of that iteration.

5.3.2 Optimistic Linear Support with Alpha Reuse

We now propose our main contribution for MOPOMDPs: *optimistic linear support with alpha reuse (OLSAR)*. This algorithm employs the OLS schema with reuse (Al-

Algorithm 13: `Perseus`$(\mathcal{A}, B, \eta)$

$\mathcal{A}' \leftarrow \mathcal{A}$;
$\mathcal{A} \leftarrow \{-\vec{\infty}\}$; // `worst possible vector in a singleton set`
while $\max_b \max_{\alpha' \in \mathcal{A}'} b \cdot \alpha' - (\max_{\alpha \in \mathcal{A}} b \cdot \alpha) > \eta$ **do**
  $\mathcal{A} \leftarrow \mathcal{A}'$; $\mathcal{A}' \leftarrow \emptyset$; $B' \leftarrow B$;
  while $B' \neq \emptyset$ **do**
    Randomly select b from B';
    $\alpha \leftarrow$ `backup`$(\mathcal{A}, b)$;
    $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{ \underset{\alpha' \in (\mathcal{A} \cup \{\mathbf{A}\})}{\arg\max} \; b \cdot \alpha' \}$;
    $B' \leftarrow \{b \in B' : \max_{\alpha' \in \mathcal{A}'} b \cdot \alpha' < \max_{\alpha \in \mathcal{A}} b \cdot \alpha\}$; // `Continue with only those` $b \in B'$ `for which the value has not yet improved`
  end
end
return $\mathcal{A}'$;

gorithm 8). Specifically, it reuses the multi-objective value of the policies, i.e., the sets $\mathcal{A}$, found in earlier calls to the single-objective subroutine. In order to do so however, we first need to define how we represent the multi-objective value as a function of the belief.

A MOPOMDP is a POMDP with a vector-valued reward function $\mathbf{R}$ instead of a scalar one. The value of a MOPOMDP policy given an initial belief b_0 is thus also a vector $\mathbf{V}_{b_0}$. The scalarized value given $\mathbf{w}$ is then $\mathbf{w} \cdot \mathbf{V}_{b_0}$. When we look at the α-vectors for a MOPOMDP, each element of an α-vector, i.e., each $\alpha(s)$, is itself a vector, indicating the value in all objectives. Thus, each α-vector is actually an α-matrix, $\mathbf{A}$, in which each row $\mathbf{A}(s)$ represents the multi-objective value vector for state s. The multi-objective value of taking the action associated with $\mathbf{A}$ under belief b (provided as a row vector) is then $b\mathbf{A}$. When $\mathbf{w}$ is also given (as a column vector), the scalarized value of taking the action associated with $\mathbf{A}$ under belief b is $b\mathbf{A}\mathbf{w}$.

Given a set of α-matrices $\mathcal{A}$ that approximates the multi-objective value function, we can thus extract the scalar value given a belief b for every $\mathbf{w}$:

$$V_b(\mathbf{w}) = \max_{\mathbf{A} \in \mathcal{A}} \; b\mathbf{A}\mathbf{w}. \tag{5.6}$$

Because each α-matrix is tagged with a certain action, a policy $\pi_{\mathcal{A}}^{\mathbf{w}}$ can be distilled from $\mathcal{A}$ given $\mathbf{w}$. We denote the multi-objective value for a given b_0 under policy π as $\mathbf{V}_{b_0}^{\pi} = b_0\mathbf{A}$.

Now that we have defined the representation of the (multi-objective) value in terms of α-matrices, we can define OLSAR. We first describe OLSAR abstractly and specify

the criteria that the single-objective POMDP method that OLSAR calls as a subroutine, which we call `solveScalarizedPOMDP`, must satisfy to be OLSAR-compliant. Then, we describe an instantiation of `solveScalarizedPOMDP` that we call *OLSAR-Compliant Perseus*. Finally, we discuss OLSAR's theoretical properties, which leverage existing theoretical guarantees of point-based methods and those of OLS (as given in Chapter 3).

Algorithm

OLSAR implements the OLS algorithm with reuse for MOPOMDPs. It thus maintains an approximate CCS, $\mathcal{S}$, and thereby an approximate scalarized value function $V^*_{\mathcal{S}}$. The vectors $\mathbf{V}_{b_0}$ in $\mathcal{S}$ are computed using sets of α-matrices. OLSAR finds these sets of α-matrices by solving a series of scalarized problems, each of which is a POMDP over the belief space for a different weight $\mathbf{w}$. Each scalarized problem is solved by a single-objective solver we call `solveScalarizedPOMDP`, which computes the value function in terms of α-matrices of the MOPOMDP scalarized by $\mathbf{w}$.

OLSAR, given in Algorithm 14, takes an initial belief b_0 and a convergence threshold η as input. The approximate CCS, and the priority queue of corner weights are computed as in Algorithm 8, in Chapter 3. The reuse of the α-matrices in OLSAR is implemented by maintaining $\mathcal{A}_{all}$: a set of all α-matrices. These α-matrices are returned by calls to `solveScalarizedPOMDP` to reuse (line 5) and B, a set of sampled beliefs (line 6).

Following the OLS schema, OLSAR repeatedly pops a corner weight off the queue. For each popped $\mathbf{w}$, it selects the α-matrices from $\mathcal{A}_{all}$ to initialize $\mathcal{A}_r$, a set of α-matrices that form a lower bound on the (multi-objective) value (line 9). Initially, this lower bound is typically an admissible heuristic. In the single-objective case, this usually consists of a minimally realizable value heuristic, in the form of α-vectors. In order to enable α-reuse for the multi-objective version, these heuristics must be in the form of α-matrices. For example, if we denote the minimal reward for each objective i as R^i_{min}, one lower bound heuristic α-matrix $\mathbf{A}_{min}$ is the vectors consisting of $R^i_{min}/(1-\gamma)$ for each objective and state.

OLSAR selects the maximizing α-matrix for each belief $b \in B$ and the given $\mathbf{w}$ and puts them in a set $\mathcal{A}_r$. Using $\mathcal{A}_r$, OLSAR calls `solveScalarizedPOMDP`. After obtaining a new set of α-matrices $\mathcal{A}_{\mathbf{w}}$ from `solveScalarizedPOMDP`, OLSAR calculates $\mathbf{V}_{b_0}$; the maximizing multi-objective value vector for b_0 at $\mathbf{w}$ (line 11). If $\mathbf{V}_{b_0}$ is an improvement to $\mathcal{S}$ (line 14), OLSAR adds it to $\mathcal{S}$ and calculates the new corner weights and their priorities.

A key insight behind OLSAR is that if we can retrieve the α-*matrices* underlying the policy found by `solveScalarizedPOMDP` for a specific $\mathbf{w}$, we can reuse these α-matrices as a lower bound for subsequent calls to `solveScalarizedPOMDP` with another weight $\mathbf{w}'$. Especially when $\mathbf{w}$ and $\mathbf{w}'$ are similar, we expect this lower bound to be close to the α-matrices required for $\mathbf{w}'$.

The corner weights that OLS selects lie increasingly closer together as the algo-

Algorithm 14: `OLSAR`(b_0, η)

$\mathcal{S} \leftarrow \emptyset$; // `partial CCS of multi-objective value vectors` $\mathbf{V}_{b_0}$
$WV_{old} \leftarrow \emptyset$; // `searched weights and scalarized values`
$Q \leftarrow$ priority queue with weights to search;
Add extrema of the weight simplex to Q with infinite priority;
$\mathcal{A}_{all} \leftarrow$ admissible heuristic lower bound in the form of α-matrices ;
$B \leftarrow$ set of sampled belief points (e.g., by random exploration);
while $\neg Q$.`isEmpty`$() \wedge \neg$*timeOut* **do**
  $\mathbf{w} \leftarrow Q$.`dequeue`(); // `Retrieve a weight vector`
  $\mathcal{A}_r \leftarrow \{\mathbf{A} : \mathbf{A} \in \mathcal{A}_{all} \wedge \exists b \in B \; b\mathbf{A}\mathbf{w} \geq \max_{\mathbf{A}' \in \mathcal{A}_{all}} b\mathbf{A}'\mathbf{w}\}$; // `the best` α`-matrices found in earlier iterations for each` $b \in B$ `for` $\mathbf{w}$
  $\mathcal{A}_{\mathbf{w}} \leftarrow$ `solveScalarizedPOMDP`$(\mathcal{A}_r, B, \mathbf{w}, \eta)$;
  $\mathbf{V}_{b_0} \leftarrow \max_{\mathbf{A} \in \mathcal{A}_{\mathbf{w}}} b_0 \mathbf{A}\mathbf{w}$;
  $\mathcal{A}_{all} \leftarrow \mathcal{A}_{all} \cup \mathcal{A}_{\mathbf{w}}$;
  $WV_{old} = WV_{old} \cup \{(\mathbf{w}, \; \mathbf{w} \cdot \mathbf{V}_{b_0})\}$;
  if $\mathbf{V}_{b_0} \notin \mathcal{S}$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{V}_{b_0}\}$;
    $W \leftarrow$ compute new corner weights and maximum possible improvements $(\mathbf{w}, \Delta_{\mathbf{w}})$ using WV_{old} and $\mathcal{S}$;
    Q.`addAll`(W);
  end
end
return $\mathcal{S}$;

rithm iterates. Consequently, the policies and value functions computed for those weights lie closer together as well and `solveScalarizedPOMDP` needs less and less time to improve upon the α-matrices that begin increasingly close to their converged values. In fact, late in the execution of OLSAR, corner weights are tested that yield no value improvement, i.e., no new α-matrices are found. These calls to the single-objective point-based subroutine, `solveScalarizedPOMDP`, serve only to confirm that the CCS has indeed been found, rather than to improve it. While such confirmation runs of `solveScalarizedPOMDP` would be expensive in OLS without reuse, in OLSAR they are cheap: the already present α-matrices suffice and `solveScalarizedPOMDP` converges immediately (after one sweep over B). The α-matrix reuse can thus save a lot of runtime late in execution, and is therefore key to the efficiency of OLSAR.

However, to exploit this insight, `solveScalarizedPOMDP` must return the α-matrices explicitly, and not just the scalarized value or the single-objective α-vectors, as standard single-objective solvers do. A naive way to retrieve the α-matrices is to perform a separate policy evaluation on the policy returned by `solveScalarizedPOMDP`. However, since POMDP policy evaluation is expensive, we instead require the single-objective point-based subroutine to be *OLSAR-compliant*: i.e., to return a set of α-matrices, while computing the same scalarized value function as a single-objective

solver.

OLS-Compliant Perseus

OLSAR requires an OLS-compliant implementation of `solveScalarizedPOMDP` that returns the multi-objective value of the policy, in the form of a set of α-matrices, found for a given $\mathbf{w}$. This requires redefining the point-based backup such that it returns an α-matrix rather than an α-vector. Because the new backup operations takes place inside `solveScalarizedPOMDP`, we now perform a backup for a given b *and* $\mathbf{w}$ starting from a set of α-matrices $\mathcal{A}_k$.

As in Equation 5.3, the new backup first computes the back-projections $\mathbf{G}_i^{a,o}$ (for all a, o) of each next-stage α-matrix $\mathbf{A}_i \in \mathcal{A}_k$. However, these $\mathbf{G}_i^{a,o}$ are now matrices instead of vectors:

$$\mathbf{G}_i^{a,o}(s) = \sum_{s' \in S} O(a, s', o) T(s, a, s') \mathbf{A}_i(s'). \tag{5.7}$$

For the given b and $\mathbf{w}$, the back-projected matrices are used to construct $|A|$ new α-matrices (one for each action):

$$\mathbf{A}_{k+1}^{b,a} = r^a + \gamma \sum_{o \in \Omega} \arg\max_{G^{a,o}} b \, \mathbf{G}^{a,o}\mathbf{w}. \tag{5.8}$$

Note that the vectors $\mathbf{G}^{a,o}\mathbf{w}$ can also be shared between all $b \in B$. Therefore, we cache the values of $\mathbf{G}^{a,o}\mathbf{w}$.

Finally, the $\mathbf{A}_{k+1}^{b,a}$ that maximizes the *scalarized value* given b and $\mathbf{w}$ is selected by the `backupMO` operator:

$$\texttt{backupMO}(\mathcal{A}_k, b, \mathbf{w}) = \arg\max_{\mathbf{A}_{k+1}^{b,a}} b \mathbf{A}_{k+1}^{a,b} \mathbf{w}. \tag{5.9}$$

We can plug `backupMO` into any point-based method. In this dissertation, we use Perseus because it is fast and can handle large sampled belief sets. The resulting OLS-compliant Perseus is given in Algorithm 15. Note that this algorithm has the same structure as Algorithm 13; the only differences are that the backups are now performed given a b *and* $\mathbf{w}$, the scalarized values are computed during Perseus are also computed given b *and* $\mathbf{w}$, and the resulting set $\mathcal{A}'$ is a set of α-matrices rather than α-vectors.

Using `solveScalarizedPOMDP` $=$ `OCPerseus` inside OLSAR yields a complete algorithm that computes an approximate CCS for MOPOMDPs. Note however that `OCPerseus` could be replaced by another OLS-compliant point planning method as well.

Analysis

Point-based methods like Perseus have guarantees on the quality of the approximation. These guarantees depend on the density δ_B of the sampled belief set, i.e., the maximal distance from the closest $b \in B$ to any other belief in the belief set (Pineau et al., 2006).

Algorithm 15: `OCPerseus`$(\mathcal{A}, B, \mathbf{w}, \eta)$

```
1  A' ← A;
2  A ← {−∞⃗};
3  while max_b max_{A'∈A'} bA'w − (max_{A∈A} bAw) > η do
4  |   A ← A'; A' ← ∅ ; B' ← B;
5  |   while B' ≠ ∅ do
6  |   |   Randomly select b from B';
7  |   |   A ← backupMO(A, b, w);
8  |   |   A' ← A' ∪ { arg max_{A'∈(A∪{A})} bA'w};
9  |   |   B' ← {b ∈ B' : max_{A'∈A'} bA'w < max_{A∈A} bAw};
10 |   end
11 end
12 return A';
```

Lemma 1. *(Pineau et al., 2006) The error ε on the lower bound of the value of an infinite-horizon POMDP after convergence of point-based methods is:*

$$\varepsilon \leq \frac{\delta_B(R_{max} - R_{min})}{(1-\gamma)^2}, \tag{5.10}$$

where R_{max} and R_{min} are the maximal and minimal possible immediate rewards.

Using Lemma 1 we can bound the error of the approximate CCS computed by OLSAR.[7]

Corollary 13. *OLSAR implemented with* `OCPerseus` *using belief set B converges in a finite number of iterations to an ε-CCS, where $\varepsilon \leq \frac{\delta_B(R_{max}-R_{min})}{(1-\gamma)^2}$.*

Proof. This follows directly from Theorem 8 and Lemma 1. □

A nice property of OLS is that it inherits any quality guarantees of the single-objective solver it uses as a subroutine. Better initialization of `OCPerseus` due to α-matrix reuse does not effect the guarantees in any way. On the contrary, it affects only empirical runtimes. The next subsection presents experimental results that measure these runtimes.

5.3.3 Experiments: MOPOMDPs

In this subsection, we empirically compare OLSAR to three baseline algorithms. The first is random sampling (RS), described in Section 3.1.2, which does not use OLS

[7]Note that because ε is constant given a set of sampled beliefs B, we can use $\mathbf{w} \cdot \mathbf{V}_{b_0}$ instead of $\mathbf{w} \cdot \mathbf{V}_{b_0} + \varepsilon$ on line 13 of Algorithm 14, because the relative priorities between corner weights in the priority queue do not change by subtracting a constant.

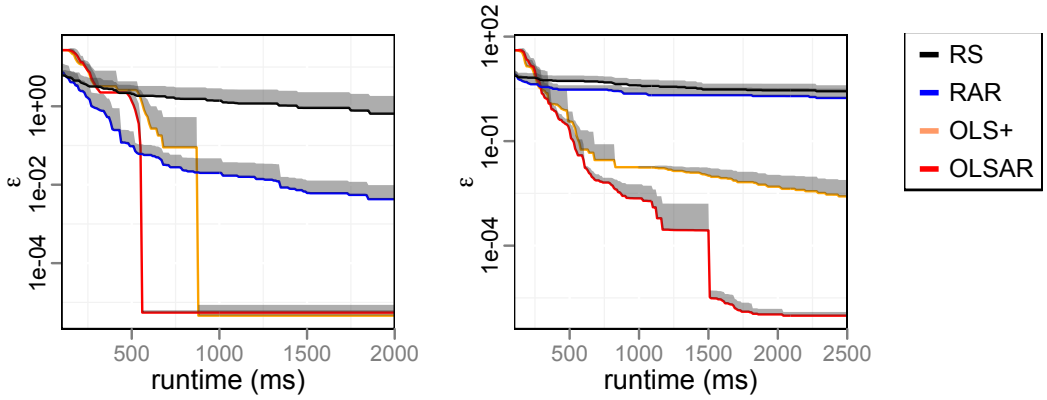


Figure 5.3: The error with respect to a reference set as a function of the runtime for (left) Tiger 2 and (right) Tiger3. The shaded regions represent standard error. Note the log scale in the y-axis. In order to avoid clutter in the plot (due to the log-scale) we only show the standard error above the lines.

or α-matrix reuse. The second is *random sampling with alpha reuse (RAR)*, which does not use OLS. The third is OLS+OCPerseus, which does not use α-matrix reuse. We do not present results for the augmented-state method of (White and Kim, 1980), as this it proved to be prohibitively computationally expensive for even the smallest problems. We tested the algorithms on three MOPOMDPs based on the Tiger (Cassandra et al., 1994) and Maze20 (Hauskrecht, 2000) benchmark POMDPs. Because we use infinite-horizon MOPOMDPs – which are undecidable – we cannot obtain the true CCS. Therefore, we compare our algorithms' solutions to *reference sets* obtained by running OLSAR with many more sampled belief points and η set 10 times smaller.

Multi-Objective Tiger

In the Tiger problem (Cassandra et al., 1994), an agent faces two doors: behind one is a tiger and behind the other is treasure. The agent can *listen* or *open* one of the doors. When it listens, it hears the tiger behind one of the doors. This observation is accurate with probability 0.85. Finding the treasure, finding the tiger, and listening yield rewards of 10, 100, and -1, respectively. We use a discount factor of $\gamma = 0.9$.

While the single-objective Tiger problem assumes all three rewards are on the same scale, it is actually difficult in practice to quantify the trade-offs between, e.g., risking an encounter with the tiger and acquiring treasure. In *two-objective MO-Tiger (Tiger2)*, we assume that the treasure and the cost of listening are on the same scale but treat avoiding the tiger as a separate objective. In *three-objective MO-Tiger (Tiger3)*, we treat all three forms of reward as separate objectives.

We conducted 25 runs of each method on both Tiger2 and Tiger3. We ran all algorithms with 100 belief points generated by random exploration, $\eta = 1 \times 10^6$, and b_0 set to a uniform distribution. The reference set was obtained using 250 belief points.

On Tiger2 (Figure 5.3 (left)), OLS+OCPerseus converges in $0.87s$ on average, with a maximal error in scalarized value across the weight simplex w.r.t. the reference set of 5×10^{-6}. OLSAR converges significantly faster (t-test: $p < 0.01$), in on average $0.53s$, with similar error 4×10^{-6} (this difference in error is not significant). RAR does not converge as it just keeps sampling new weights. However, we can measure error w.r.t. the reference set. After $2s$ (about four times what OLSAR needs to converge), RAR has an error with respect to the reference set of 4.5×10^{-3} (a factor of 10^3 higher than OLSAR after convergence). RS is even worse and does not come further than a 0.78 error after $2s$. OLS+OCPerseus and OLSAR perform similarly in the first half: they use about the same time on the first four iterations ($0.35s$). However, OLSAR is significantly faster for the second half. Thus, on Tiger2, OLS-based methods are significantly better than random sampling and α-reuse significantly speeds the discovery of good value vectors.

The results for the Tiger3 problem are in Figure 5.3 (right). RS and RAR both perform poorly, with RAR better by a factor of two. OLS+Perseus is faster than RAR but OLSAR is the fastest and converges in about $2s$. OLS+OCPerseus eventually converges to the same error but takes 10 times longer. As before, OLS-based methods outperform random sampling. Furthermore, for this 3-objective problem, α-reuse speeds convergence of OLSAR by an order of magnitude.

We also compared three variations of OLSAR: using all previously found α-matrices, only reusing those found for the policy of the closest previous weight, and using those α-matrices for policies whose multi-objective values make up the current corner weight. However, no significant differences where found between their convergence times or final error, neither in the 2- nor in the 3-objective Tiger problem.

Multi-Objective Maze20

In Maze20 (Hauskrecht, 2000), a robot must navigate through a 20-state maze. It has four actions to move in the cardinal directions, plus two sensing actions to perceive walls in the north-south or east-west directions. Transitions and observations are stochastic. In the single-objective problem, the agent gets reward of 2 for sensing, 4 for moving while avoiding the wall, and 150 for reaching the goal. In *multi-objective Maze 20 (MO-Maze20)*, the reward for reaching the goal is treated as a separate objective, yielding a two-objective problem.

To test performance of OLSAR on MO-Maze20, we first created a reference set by letting OLSAR converge using 1500 sampled beliefs. This took 11 hours. Then we ran OLSAR, OLS+OCPerseus, RAR, and RS with 1000 sampled beliefs. Due to the computational expense, we conducted only three runs per method. Figure 5.4 shows the results. OLSAR converges after (on average) 291 minutes ($4.90hrs$) at an error of 0.09, less than 0.5% w.r.t. the reference set. We let OLS+OCPerseus run for 11 hours but it did not converge in that time. OLS+OCPerseus, RS, and RAR have similar performance after around 300 minutes. However, until then, OLS+OCPerseus does better (and unlike RS and RAR, OLS+OCPerseus is guaranteed to converge). OLSAR

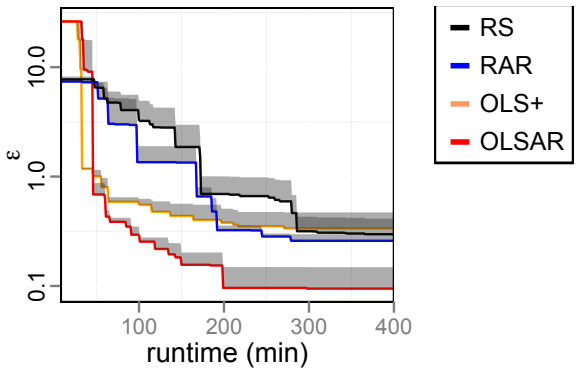


Figure 5.4: The error with respect to a reference set as a function of the runtime for MO-Maze20. The shaded regions represent standard error. Note the log scale in the y-axis. In order to avoid clutter in the plot (due to the log-scale) we only show the standard error above the lines.

converges at significantly less error than what the other methods have attained after 400 minutes. We therefore conclude that OLSAR reduces error more quickly than the other algorithms and converges faster than OLS+OCPerseus.

5.4 Conclusion

In this chapter we have proposed novel methods for multi-objective sequential decision problems, i.e., MOMDPs and MOPOMDPs, using the OLS framework of Chapter 3. First, we tackled a large MOMDP called the *maintenance planning problem (MPP)* using OLS-based methods. Second, we proposed a new method for infinite-horizon MOPOMDPs, that — as far as we are aware — is the first method to reasonably scale.

For the MPP, we have shown that it is possible to create multi-objective methods on the basis of OLS and state-of-the single-objective subroutines that are tailored for the MPP. We have shown that it is easy to replace these subroutines if the state-of-the-art for a given problem improves. We compared exact CCS planning methods, i.e., OLS+SPUDD and OLS+CoRe, to an approximate planning method, i.e., OLS+UCT*. We showed that while the approximate subroutine does not provide bounds on the quality of approximation, OLS+UCT* can quickly produce approximate CCSs with low error, ε, in practice. We therefore conclude that the OLS framework offers a flexible framework that is easy to use for large problems, especially when solvers for the scalarized, i.e., single-objective, versions of these problems already exist.

For MOPOMDPs, we proposed, analyzed, and tested OLSAR, a novel algorithm for MOPOMDPs that intelligently selects a sequence of scalarized POMDPs, by building on the OLS framework. OLSAR uses OCPerseus, a scalarized MOPOMDP solver that returns the multi-objective value of the policies it finds, as well as the α-matrices

that describe them. A key insight underlying OLSAR is that these α-matrices can be reused in subsequent calls to OCPerseus, greatly reducing runtimes in practice. Furthermore, because OCPerseus returns ε-optimal policies, OLSAR is guaranteed to return an ε-CCS. Our experiments show that OLSAR greatly outperforms alternatives that do not use OLS and/or α-matrix reuse. We therefore conclude that OLS *and* reuse are key to keeping MOPOMDPs tractable.

Chapter 6

Conclusion

In this dissertation, we aimed to answer the following question: *"Can we create fast multi-objective planning algorithms for cooperative decision problems that are: either single- or multi-agent, single-shot or sequential, and fully or partially observable?"* To answer this question affirmatively, we have discussed several models and proposed algorithms for multi-objective decision-theoretic planning. In this final chapter, we first reiterate our contributions and discuss their implications for the field of multi-objective decision theory in Section 6.1. In Section 6.2, we critically review our own work and outline several possibilities for future research.

6.1 Contributions

In this section we summarize our contributions. Our contributions are two-fold; we have made contributions that apply to cooperative multi-objective decision problems in general, and we have made problem-specific contributions. We start with the big picture, and then go into the problem-specific contributions. For our contributions, we briefly discuss what we think the implications are for future research in this research direction.

6.1.1 The big picture

The research we presented in this dissertation is motivated by the need for solving decision-problems with multiple objectives; not only are there many real-world problems that have multiple objectives, but we have also argued that these often cannot be solved with existing single-objective techniques, because it is often impossible to *a priori* scalarize, i.e., convert multi-objective problems to single-objective ones.

In order to make explicit under what circumstances special methods are needed to solve multi-objective problems, we identified three scenarios in which it is impossible, infeasible, or undesirable to scalarize a priori in Chapter 1. As well as providing mo-

tivation for the need for multi-objective methods, these scenarios also represent three main ways these methods are applied in practice.

We proposed a taxonomy that classifies multi-objective methods according to the applicable scenario, the scalarization function (which projects multi-objective values to scalar ones), and the type of policies that are allowed in Chapter 2. We showed how these three aspects together determine the nature of an optimal solution, which can be a single policy, or a coverage set containing multiple policies. We limited our scope to problems for which multiple policies are required.

Our taxonomy is based on a *utility-based approach*, which sees the scalarization function as part of the utility, and thus part of the problem definition. This contrasts with the so-called *axiomatic approach*, which usually assumes the Pareto front is the appropriate solution. We showed that the utility-based approach can be used to justify the choice for a solution set. Based on the utility-based approach we argued for a particular solution set: the *convex coverage set (CCS)*, which applies to two important parts of the taxonomy. The first is the case in which the scalarization function is linear, i.e., the inner product of a vector of positive weights, $\mathbf{w}$, and the vector-valued value of a policy in a multi-objective decision problem, and the second is when the scalarization function is monotonically increasing in all objectives and stochastic policies are allowed. The CCS is the central solution concept of this dissertation, and we propose several novel planning algorithms for determining the CCS in a variety of multi-objective cooperative planning problems.

In Chapter 3, we discussed the two main approaches to create new CCS algorithms building from an existing single-objective planning algorithm. The first approach, that is often taken in the literature, we refer to as the inner loop approach. In this approach, the maximizations and summations — that are necessary to compute the optimal policy and policy value of a single-objective decision problem — inside the inner most loops of the single-objective algorithms are replaced with appropriate pruning operators and cross-sums between sets of value vectors. Inner loop algorithms are as problem-specific as the single-objective algorithms on which they are based. The second approach is the outer loop approach, in which a series of scalarized problems are solved using the single-objective method as a subroutine in order to incrementally build up the CCS for a multi-objective problem. Because these single-objective subroutines can be left unchanged, outer loop methods are generic frameworks: they can apply to any (scalarizable) multi-objective decision problem. Important disadvantages of existing outer loop methods however, include that they often do not converge to the exact CCS, except for in the limit, and that there are no quality bounds for the intermediate approximate CCSs. Therefore, we propose a new outer loop method: the *optimistic linear support (OLS)* framework.

The OLS framework takes inspiration from the POMDP literature (and specifically Cheng's linear support algorithm (Cheng, 1988)), to exploit the piecewise linearity and convexity of the scalarized value function (under linear scalarization) in multi-objective decision problems. OLS is an outer loop approach, and thus solves a series of scalarized problems, for different (linear) scalarization weights $\mathbf{w}$. At each itera-

tion, OLS identifies the single weight vector $\mathbf{w}$ that can lead to the maximal possible improvement on a partial CCS it has already identified, and calls a single-objective subroutine to solve a scalarized instance of the problem for $\mathbf{w}$. By doing so, OLS is guaranteed to terminate in a finite number of iterations. Furthermore, when the single-objective subroutine is exact, i.e., guaranteed to find the optimal policy for the scalarized instance of the problem, OLS is guaranteed to return the exact CCS. Before termination, OLS can bound the quality loss if the partial CCS it has identified so far would be used to approximate the exact CCS, in terms of lost scalarized value. In other words, it is an *anytime* algorithm, for which each intermediate CCS is an ε-CCS. We have shown empirically (for MO-CoGs in Chapter 4) that it is often possible to produce an ε-CCS in a fraction of the time it takes to produce an exact CCS, even for small ε.

When exact single-objective subroutines are not available, OLS can also function in combination with approximate single-objective subroutines. When such an approximate subroutine produces a bounded approximation with at most ε error, OLS is guaranteed to produce an ε-CCS upon termination. We have provided practical examples of this in the context of MO-CoGs (Chapter 4) and MOPOMDPs (Chapter 5).

Often, OLS can be further improved by reusing the policies, value functions, or other artifacts produced by the single-objective subroutines during earlier iterations of OLS to hot-start the single-objective subroutines in later iterations. The precise information that can be reused depends on the problem. We have shown the reuse of a representation of the value function in the form of α-matrices in MOPOMDPs to be particularly effective in Chapter 5.

When we compare inner loop algorithms and OLS-based algorithms that use the same single-objective methods as a basis/subroutine, we observe that inner loop algorithms typically scale better in the number of objective of a multi-objective decision problem, and OLS-based algorithms scale better in the size of the scalarized problems (such as the number of agents in a (MO-)CoG). We therefore conclude that inner loop and outer loop methods are complementary in this respect. However, the inner loop methods are typically not anytime, and need to run until completion before producing a result. Additionally, contrary to inner loop methods, OLS imposes very little additional memory demands in addition to the memory demands of the single-objective subroutines it employs. This can be a deciding factor when memory is limited, as we have shown in the context of MO-CoGs in Chapter 4. Furthermore, while it is relatively easy to replace a subroutine in OLS (as we show for example in Chapter 5 in the context of large MOMDPs), it is typically far from trivial to create a new inner loop method based on a different single-objective algorithm.

We conclude that OLS is a framework that can be applied in the context of different multi-objective decision problems (with relatively little effort if a single-objective subroutine is available), is faster than corresponding inner loop methods for small and medium objectives, and typically uses less memory than inner loop methods.

We believe that the advantages of OLS can be exploited in many future CCS planning methods. Especially when real-world problems require the usage of problem-specific solution methods, we argue that OLS provides a fast and relatively low-effort

option for creating a multi-objective solution method.

6.1.2 Problem-specific contributions

In this dissertation, we have made contributions to three classes of decision problems: *multi-objective coordination graphs (MO-CoGs)*, *multi-objective Markov decision processes (MOMDPs)* and *multi-objective partially observable Markov decision processes (MOPOMDPs)*.

Multi-objective coordination graphs

For MO-CoGs, we have proposed five novel algorithms: *convex multi-objective variable elimination (CMOVE)*, *convex AND/OR tree search (CTS)*, *variable elimination linear support (VELS)*, *AND/OR tree search linear support (TSLS)* and *variational optimistic linear support (VOLS)*. The first two are inner loop methods, while the last three are outer loop methods that build on *optimistic linear support (OLS)*.

We noted that it is possible to take the inner loop approach when building upon single-objective methods whose core operations consist of maximizations and summations, such as *variable elimination* and *AND/OR tree search*. However, when the core operations are different, such as for *variational methods* — in which reparameterizations and restructuring of the set of local payoff functions form the core operations — it is not clear how to take an inner loop approach. It is always possible to take an outer loop approach for MO-CoGs, as it can use a complete and unchanged single-objective solver as a subroutine, and evaluate the multi-objective value of joint actions that are produced by this subroutine. An example of this is our VOLS algorithm, which builds on OLS and employs variational methods as subroutines. Because OLS inherits both quality and memory guarantees from the single-objective subroutines it employs, OLS opens up the way to creating multi-objective methods with favorable quality and memory properties from single-objective methods to which an inner loop approach does not apply. We consider this an important advantage in the context of MO-CoGs as its scalarized version, i.e., the single-objective CoG[1], is a large and active field of research that has produced many such methods. Furthermore, because it takes relatively little effort to plug in a new subroutine into OLS, it offers an attractive framework to create state-of-the-art multi-objective methods.

We compared the inner loop methods we propose to their corresponding OLS-based methods. Specifically, we compared CMOVE to VELS and CTS to TSLS. We showed both theoretically and empirically that OLS-based methods scale better in the number of agents in terms of runtime, while inner loop methods scale better in the number of objectives. We showed — both theoretically and empirically — that OLS-based methods perform better overall in terms of memory requirements than inner loop methods.

[1] Or equivalents of the CoG, such as finding the MAP in probabilistic graphical models (Pearl, 1988).

We have shown that OLS-based algorithms are anytime, and that any intermediate solution set during the execution of an OLS-based algorithm is an ε-CCS, as long as the quality bounds of the single-objective method are known. We have used this result to show empirically, e.g., using VELS, that we can produce ε-CCSs in a fraction of the runtime that is required to produce an exact CCS. Furthermore, when the single-objective subroutine, such as the variational subroutine in VOLS, is not exact, but does produce an ε-approximate solution, that OLS inherits this quality guarantee and can produce an ε-CCS. This is a major advantage, as (MO-)CoGs are NP-hard, and therefore most state-of-the-art solvers are approximate.

We have shown with VOLS, that it is possible to reuse parts of the solutions/artifacts produced by single-objective variational subroutines in earlier iterations of OLS, in order to hot-start the variational subroutine in later iterations. Specifically, VOLS reuses the restructured and reparameterized set of local payoff functions that variational methods produce. We have shown empirically that this can lead to better runtime and/or quality results.

When we compare our outer and inner loop methods in terms of runtime, memory usage, and quality guarantees, we note that exact methods (CMOVE, CTS, VELS and TSLS) have runtimes that depend exponentially on the induced width. Of these exact methods, VELS is typically fastest and when the number of objectives is low or medium, and TSLS is most memory-efficient. For higher number of objectives, inner loop methods are typically faster than outer loop methods; CMOVE scales better than VELS in the number of objectives, but has poor memory guarantees; CTS has better memory usage than CMOVE but is slower. OLS-based outer loop methods are always more memory-efficient than their inner loop counterparts. Compared to exact methods, VOLS scales much better in the number of agents, i.e., the runtime does not increase exponentially with the number of agents. However, VOLS only produces an ε-CCS.

Summarizing, we have proposed an analyzed five novel algorithms for MO-CoGs that compute a (approximate) CCS. Our methods offer different trade-offs between runtime, memory usage, and quality guarantees.

Multi-objective MDPs

For MOMDPs, we have demonstrated the efficacy of OLS when the MOMDP is very large. In order to illustrate this we have made use of the maintenance planning problem (MPP) (Scharpff et al., 2013; Roijers et al., 2014a; Scharpff et al., 2015, 2016). The MPP is notoriously hard due to its large state and action space. Therefore, the existing methods for the single-objective version of the MPP exploit the specific problem structure of the MPP; the SPUDD and UCT* algorithms employ a problem-specific encoding (Scharpff et al., 2013) while the CoRe algorithm (Scharpff et al., 2016) — that was proposed only recently — exploits the transition independence between features of the state-space.

We have shown that it possible to create planning methods that compute the CCS for problems like the MPP by using state-of-the-art, possibly problem-specific, single-

objective solvers as subroutines inside OLS. Because it is relatively easy to exchange subroutines (but not to create inner loop methods from new single-objective methods), we conclude that especially for problems that require a problem-specific approach, OLS is key to the creation of effective multi-objective planning methods.

Multi-objective POMDPs

In this dissertation, we proposed, analyzed, and tested *optimistic linear support with alpha reuse (OLSAR)*. Because OLSAR uses the OLS framework, it intelligently selects a sequence of scalarized POMDPs to solve in order to solve MOPOMDPs. As far was we are aware, this is the first MOPOMDP planning method that computes the CCS and reasonably scales in the number of states of the MOPOMDP.

We have shown how to represent the value function (in the belief b and the scalarization weight $\mathbf{w}$) in terms of α-matrices. OLSAR uses this representation. The single-objective subroutine we propose for OLSAR is *OLS-compliant Perseus (OCPerseus)*, a scalarized MOPOMDP solver that returns the multi-objective value of the policies it finds, as well as the α-matrices that describe them. A key insight underlying OLSAR is that the α-matrices produced by OCPerseus can be reused in subsequent calls to OCPerseus. This α-reuse greatly reduces runtimes in practice. Furthermore, since OCPerseus returns ε-optimal policies, OLSAR is guaranteed in turn to return an ε-CCS. Our experimental results show that OLSAR greatly outperforms alternatives that do not use OLS and/or α-matrix reuse.

6.2 Discussion and Future Work

In this section, we critically review the work in this dissertation, and identify opportunities for future work.

6.2.1 On the sufficiency of the taxonomy

In Chapter 2, we introduced a taxonomy based on the utility-based approach. We assumed the existence of a *utility* or *scalarization* function, f.

In this dissertation we have restricted f to monotonically increasing functions in all objectives. We argue that this is a minimal assumption because it simply states that increasing the value with respect to one objective without diminishing the others, cannot decrease the scalarized value. This — in our view — corresponds to what we mean by 'objective'. However, it is possible to have a preference ordering over value vectors, without there being an f that corresponds to it. When preference is based on a lexicographical ordering, i.e., even an infinitesimal improvement in an objective with a higher priority is better than a large improvement in an objective with a lower priority, and the rewards are vectors in $\Re^n$, there is no $f : \Re^n \rightarrow \Re$ that corresponds to this

(Sen, 1995). Therefore, lexicographical orderings falls outside of our mathematical formalism.

We have used one special case of f, namely the linear (and monotonically increasing) scalarization function. However, there are more special cases of f corresponding to different preference orderings described in the literature. For example, when certain "fairness" constraints are imposed on f this leads to the Lorenz front (Perny et al., 2013), which is a subset of the Pareto front. We have not identified this, or other, additional constraints that can be imposed on f. However, it would be useful for future work in multi-objective decision making to have an overview of such constraints, and especially in what type of situations they occur.

6.2.2 Scalarized expectation of return versus expectation of scalarized return

In Section 1.1 in Definition 1, we defined the scalarized value of a policy, π, to be the result of applying the scalarization function f to the multi-objective value $\mathbf{V}^\pi$, i.e., $V^\pi_\mathbf{w}(s) = f(\mathbf{V}^\pi(s), \mathbf{w})$. Since $\mathbf{V}^\pi(s)$ is typically itself an expectation, this means that the scalarization function is applied *after* the expectation is computed, e.g., for MOMDPs (Definition 33):

$$V^\pi_\mathbf{w}(s) = f(\mathbf{V}^\pi(s), \mathbf{w}) = f(E[\sum_{k=0}^{\infty} \gamma^k \mathbf{r}_k \mid \pi, s_0 = s], \mathbf{w}).$$

This formulation, which we refer to as the *scalarization of the expected return (SER)* is standard in the literature. However, it is not the only conceivable option. It is also possible to define $V^\pi_\mathbf{w}(s)$ as the *expectation of the scalarized return (ESR)*, e.g., for MOMDPs:

$$V^\pi_\mathbf{w}(s) = E[f(\sum_{k=0}^{\infty} \gamma^k \mathbf{r}_k, \mathbf{w}) \mid \pi, s_0 = s]$$

Which definition is used can critically affect which policies are preferred. For example, consider the following 2-objective infinite-horizon MOMDP, illustrated in Figure 6.1. There are four states (A, B, C, and D). The agent starts in state A and has two possible actions: a_1 transits to state B or C, each with probability of 0.5, and a_2 transits to state D with probability 1. Both actions lead to a $(0, 0)$ reward. In states B, C and D there is only one action, which leads to a deterministic reward of $(3, 0)$ for B, $(0, 3)$ for C, and $(1, 1)$ for D.

The scalarization function just multiplies the two objectives together if the value in both objectives is greater than 0, and is 0 otherwise. Thus for this MOMDP with only positive rewards, the scalarized value under SER is,

$$V^\pi_\mathbf{w}(s) = V^\pi_1(s) V^\pi_2(s),$$

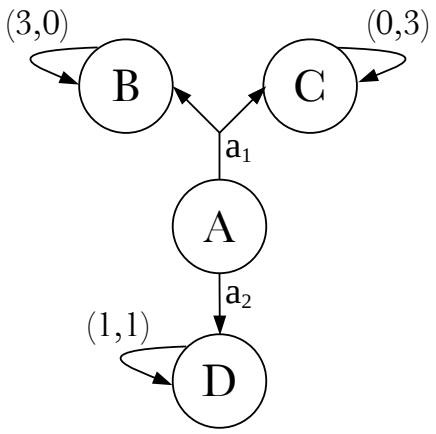


Figure 6.1: An MOMDP with two objectives and four states.

and under ESR,

$$V^{\pi}_{\mathbf{w}}(s) = E[\Big(\sum_{k=0}^{\infty} \gamma^k r^1_k\Big)\Big(\sum_{k=0}^{\infty} \gamma^k r^2_k\Big) \mid \pi, s_0 = s],$$

where r^i_k is the reward for the i-th objective on timestep k ($\mathbf{w}$ is not needed in this example since f involves no constants). If $\pi_1(A) = a_1$ and $\pi_2(A) = a_2$, then the multi-objective values are $\mathbf{V}^{\pi_1}(A) = (1.5\gamma/(1-\gamma), 1.5\gamma/(1-\gamma))$ and $\mathbf{V}^{\pi_2}(A) = (\gamma/(1-\gamma), \gamma/(1-\gamma))$.

Under SER, this leads to scalarized values of $V^{\pi_1}(A) = (1.5\gamma/(1-\gamma))^2$ and $V^{\pi_2}(A) = (\gamma/(1-\gamma))^2$ and consequently π_1 is preferred. Under ESR, however, we have $V^{\pi_1}(A) = 0$ and $V^{\pi_2}(A) = (\gamma/(1-\gamma))^2$ and thus π_2 is preferred.

Intuitively, the SER formulation is appropriate when the policy will be used many times and *return accumulates across episodes*, for example because the same user is using the policy each time. Then, scalarizing the expected reward makes sense and π_1 is preferable because in expectation it will accumulate more return in both objectives.

However, if the policy will only be used a few times or the return does not accumulate across episodes, for example because each episode is conducted for a different user, then the ESR formulation seems more appropriate. In this case, the expected return before scalarization is not of interest and π_2 is preferable because π_1 will always yield zero scalarized return on any given episode.

Another question that is related to this issue is whether or not to allow stochastic policies, i.e., when a policy will be used many times and *return accumulates across episodes*, as we argued for the SER formulation, then it also seems logical to allow stochastic policies in order to increase the scalarized expected return. When a policy will only be used a few times or the return does not accumulate across episodes, as we argued for the ESR formulation, it also seems inappropriate to allow stochastic policies.

To our knowledge, there is no literature on multi-objective decision-theoretic planning and learning that employs the ESR formulation, even though there are many real-world scenarios in which it seems more appropriate. For example, in the medical application of Lizotte et al. (2010), each patient gets only one episode to treat his or her illness, and needs to balance the effectiveness of the treatment against the severity of the side-effects. Therefore, the patient clearly interested in maximizing ESR, not SER. Thus, we believe that developing methods for the ESR formulation is a critical direction for future research in multi-objective decision making.

6.2.3 Other decision problems

For the MO-CoG, MOMDP and MOPOMDP models we propose new algorithms. However, there are more possible collaborative MODPs, that extend SODPs from Figure 2.2 that have been studied in the literature. We briefly discuss those models here, and how they are related to the previously discussed MODPs.

Firstly, *collaborative Bayesian games (CBGs)* (Oliehoek et al., 2012) are single-shot, multi-agent models like CoGs. Unlike CoGs however, each agent receives a private observation, called a type θ, before it needs to act. The distributions over types for each agent are common knowledge. Deterministic policies in a CBG specify for an action for each agent for every type of that agent. CBGs can be flattened to a large CoG by for enumerating the possible mappings from types to actions for each agent and seeing these mappings as that agent's local action space, $\mathcal{A}_i$. This same reduction could be made for a multi-objective version of a CBG to a MO-CoG. CBGs are often used as a subproblem in the sequential version of the problem, the Dec-POMDP (MacDermed and Isbell, 2013).

The class of multi-agent, sequential, and partially observable collaborative models is called the *multi-agent Markov decision process (MMDP)* model in the single-objective-setting. Similar to flattening a CoG into a BP, by treating the team of agents as one central agent with a very large action space, an MMDP can be flattened to a very large MDP, as we have mentioned in Section 5.2 for the maintenance planning problem. However, the joint state and action spaces of an MMDP are exponentially sized in the number of agents, so solving an MMDP in such a *centralized* manner, is typically infeasible. Therefore, MMDPs can — in general — only be exactly solved for small numbers of agents (e.g., Scharpff et al. (2015)). For larger MMDPs, approximate methods are required. These approximate methods — an overview of which is given by Spaan et al. (2011) — either rely on an artificial factorization of the value function (Guestrin et al., 2002), or *decentralization*, which leads to a partially observable problem setting. At present, decentralized approximation methods typically do not provide bounds with respect to the approximation quality due to decentralization. It may be an interesting research possibility to try and construct methods that do provide such bounds. Once such methods are invented, their quality bounds will immediately transfer to the multi-objective setting, when they are used as a subroutine inside OLS.

The class of multi-agent, sequential, and partially observable collaborative mod-

els is known as the *decentralized partially observable Markov decision process (Dec-POMDP)* (Bernstein et al., 2002; Oliehoek, 2010) model in the single-objective setting. Recently, major strides in solving Dec-POMDPs have been made. In particular, it has been shown that there is a reduction from Dec-POMDP to a special type of centralized POMDP called a *non-observable Markov decision process (NOMDP)* (MacDermed and Isbell, 2013; Nayyar et al., 2013; Dibangoye et al., 2013; Oliehoek and Amato, 2014). This allows POMDP solution methods to be employed in the context of Dec-POMDPs. For *multi-objective Dec-POMDPs (MO-Dec-POMDPs)*, the reduction to NOMDPs is potentially very useful as well. Specifically, an equivalent reduction from an MO-Dec-POMDP to a *multi-objective NOMDP (MONOMDP)* could be made, for which the OLSAR algorithm that we propose for MOPOMDPs in Section 5.3 would apply.

6.2.4 Other aspects of decision problems

Aside from the aspects that are discussed in Section 1.3, there are several other orthogonal aspects that are studied in the literature, such as continuous time (Van Hasselt, 2012; de Sousa Messias, 2014) and non-Markovian states (Das et al., 1999). These aspects are in principle compatible with our approach. Models in which there are multiple self-interested or even adversarial agents (Brown et al., 2012; Wiggers et al., 2015) however, are not compatible with our approach, as Assumption 1 — on which our methods rely — does not hold in these settings. It would be interesting to investigate whether an outer loop approach may still be interesting in self-interested settings by analyzing the properties of the scalarized value function of these problems.

6.2.5 Reuse in OLS and scalability in the number of objectives

As we have mentioned several times in this dissertation, OLS works well when the number of objectives is low. OLS scales poorly in the number of objectives because the number of calls to the single objective solver depends on the number of corner weights, which is exponential in the number of objectives (Theorem 5).

As we have seen in Section 5.3, when we employ reuse (Section 3.6), the effort for corner weights for which no new value is found can be drastically reduced in practice by hot-starting the single-objective subroutine. For MOPOMDPs this is even vital to keep the planning problem tractable.

In this dissertation, we have not yet made a theoretical analysis of the effect of reuse and its effect on the scalability in the number of objectives. Intuitively, when the single-objective method terminates immediately after a hot-start when no new value vector can be identified this reduces the runtime to the runtime of the single-objective solver times the number of solutions in the final CCS. In practice however, the single-objective solver will need some time to determine whether improvements can be found. Furthermore, as we have seen in VOLS in Section 4.4.6, reuse is not always guaranteed to improve the runtime for every run of the single-objective solver.

In future work, we aim to provide an analysis of the effect of reuse upon runtime — and specifically on the scalability in the number of objectives — in different OLS-based methods when it can be proven that: the single-objective solver used terminates in one iteration (as in OLSAR) or when an extra check to confirm the optimality of a partial CCS for a new $\mathbf{w}$ can be introduced, and that reuse can only improve the runtime of the single-objective solver.

6.2.6 Decision makers in the loop

In this dissertation we have assumed that we know the constraints on the scalarization function, f, produce a coverage set by running a planning algorithm, and only in a separate selection phase present that coverage set to the decision maker. There are however two issues with this workflow: first, the coverage set might be too large for a decision maker to analyze, and second, we might be doing a lot of unnecessary work because the constraints on f are too loose.

An interesting way to mitigate both issues is to involve the decision maker while planning, i.e., when we produce intermediate results during the planning phase, we might elicit more preference information from the decision maker by asking questions about these intermediate results (Benabbou and Perny, 2015). We believe that this particularly compatible with OLS, as decision makers can be presented with values of complete policies from the partial CCS, $\mathcal{S}$, between iterations of OLS. The answers that the user provides can be used by OLS to quickly reduce the interesting region of the weight space, leading to large speed ups.

Bibliography

Altman, E. (2002). Applications of markov decision processes in communication networks. In *Handbook of Markov decision processes*, pages 489–536. Springer.

Arnborg, S. (1985). Efficient algorithms for combinatorial problems on graphs with bounded decomposability — a survey. *BIT Numerical Mathematics*, 25(1):1–23.

Åström, K. J. (1965). Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205.

Auer, P. and Ortner, R. (2010). UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, 61(1-2):55–65.

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

Barrett, L. and Narayanan, S. (2008). Learning all optimal policies with multiple criteria. In *ICML 2008: Proceedings of The Twenty-fifth International Conference on Machine Learning*, pages 41–47.

Bayardo, R. J. J. and Miranker, D. P. On the space-time trade-off in solving constraint satisfaction problems. In *IJCAI 1995: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 558–562.

Becker, R., Zilberstein, S., Lesser, V., and Goldman, C. (2003). Transition-Independent Decentralized Markov Decision Processes. In *AAMAS 2003: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 41–48.

Bellman, R. (1957a). *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA.

Bellman, R. E. (1957b). A Markovian decision process. *Journal of Mathematics and Mechanics*, 6:679–684.

Benabbou, N. and Perny, P. (2015). Incremental weight elicitation for multiobjective state space search. In *AAAI 2015: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1093–1099.

Bergemann, D. and Valimaki, J. (2006). Efficient dynamic auctions. Cowles Foundation Discussion Paper.

Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of Markov decision processes. *Mathematics of operations research*, 27(4):819–840.

Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific, Nashua (NH), USA.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer, New York (NY), USA.

Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. In *TARK 1996: Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210.

Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94.

Brown, M., An, B., Kiekintveld, C., Ordóñez, F., and Tambe, M. (2012). Multi-objective optimization for security games. In *AAMAS 2012: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 863–870.

Busoniu, L., Babuska, R., and De Schutter, B. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics*, 38(2):156–172.

Cassandra, A., Littman, M., and 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*, pages 54–61.

Cassandra, A. R. (1998). *Exact and approximate algorithms for partially observable Markov decision processes*. PhD thesis, Brown University.

Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *AAAI 1994: Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1023–1028.

Cavallo, R. (2008). Efficiency and redistribution in dynamic mechanism design. In *EC 2008: Proceedings of the 9th ACM conference on electronic commerce*, pages 220–229.

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

Clemen, R. T. (1997). *Making Hard Decisions: An Introduction to Decision Analysis*. South-Western College Pub, 2 edition.

Das, T. K., Gosavi, A., Mahadevan, S., and Marchalleck, N. (1999). Solving semi-Markov decision problems using average reward reinforcement learning. *Management Science*, 45(4):560–574.

de Sousa Messias, J. V. T. (2014). *Decision-Making under Uncertainty for Real Robot Teams*. PhD thesis, Instituto Superior Técnico, Lisbon, Portugal.

Dechter, R. (1998). Bucket elimination: A unifying framework for probabilistic inference. In *Learning in graphical models*, pages 75–104. Springer Netherlands.

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

Dechter, R. and Mateescu, R. (2007). AND/OR search spaces for graphical models. *Artificial intelligence*, 171(2):73–106.

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

Dibangoye, J. S., Amato, C., Buffet, O., and Charpillet, F. (2013). Optimally solving Dec-POMDPs as continuous-state MDPs. In *IJCAI 2013: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*.

Drugan, M. M. and Nowé, A. (2013). Designing multi-objective multi-armed bandits algorithms: A study. In *IJCNN 2013: Proceedings of the 2013 International Joint Conference on Neural Networks*, pages 1–8.

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

Dubus, J., Gonzales, C., and Perny, P. (2009b). Multiobjective optimization using GAI models. In *IJCAI 2009: Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence*, pages 1902–1907.

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

Franklin, S. and Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent agents III agent theories, architectures, and languages*, pages 21–35. Springer.

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

Gonzales, C. and Perny, P. (2004). GAI networks for utility elicitation. In *AAAI: Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 224–234.

Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters*, 1(4):132–133.

Guestrin, C., Koller, D., and Parr, R. (2002). Multiagent planning with factored MDPs. In *NIPS 2002: Advances in Neural Information Processing Systems 15*, pages 1523–1530.

Guizzo, E. (2011). How Google's self-driving car works. *IEEE Spectrum Online, October*, 18.

Handa, H. (2009a). EDA-RL: Estimation of distribution algorithms for reinforcement learning problems. In *GECCO 2009: ACM/SIGEVO Genetic and Evolutionary Computation Conference*, pages 405–412.

Handa, H. (2009b). Solving multi-objective reinforcement learning problems by EDA-RL - acquisition of various strategies. In *ISDA 2009: Proceedings of the Ninth Internatonal Conference on Intelligent Sysems Design and Applications*, pages 426–431.

Hauskrecht, M. (2000). Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 13:33–94.

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

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD: Stochastic planning using decision diagrams. In *UAI 1999: Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 279–288.

Howard, R. A. (1960). *Dynamic programming and Markov decision processes*. MIT Press.

Ihler, A. T., Flerova, N., Dechter, R., and Otten, L. (2012). Join-graph based cost-shifting schemes. In *UAI 2012: Proceedings of the Twenty-Eighth Annual Conference on Uncertainty in Artificial Intelligence*, pages 397–406.

Inja, M., Kooijman, C., de Waard, M., Roijers, D. M., and Whiteson, S. (2014). Queued Pareto local search for multi-objective optimization. In *PPSN 2014: Proceedings of the Thirteenth International Conference on Parallel Problem Solving from Nature*, pages 589–599.

Jarvis, R. A. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21.

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

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

Keller, T. and Eyerich, P. (2012). PROST: Probabilistic Planning Based on UCT. In *ICAPS 2012: Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 119–127.

Keller, T. and Helmert, M. (2013). Trial-based heuristic tree search for finite horizon MDPs. In *ICAPS 2013: Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pages 135–143.

Ketter, W., Peters, M., and Collins, J. (2013). Autonomous agents in future energy markets: the 2012 power trading agent competition. In *BNAIC 2013: Proceedings of the 25th Benelux Conference on Artificial Intelligence*, pages 328–329.

Kim, I. Y. and de Weck, O. L. (2005). Adaptive weighted-sum method for bi-objective optimization: Pareto front generation. *Structural and multidisciplinary optimization*, 29(2):149–158.

Kober, J. and Peters, J. (2012). Reinforcement learning in robotics: A survey. In Wiering, M. A. and Van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 579–610. Springer Berlin/Heidelberg.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282–293.

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

Kok, J. R. and Vlassis, N. (2004). Sparse cooperative Q-learning. In *ICML 2004: Proceedings of the twenty-first international conference on Machine learning*, pages 61–68.

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

Kooijman, C., de Waard, M., Inja, M., Roijers, D. M., and Whiteson, S. (2015). Pareto local policy search for MOMDP planning. In *ESANN 2015: Special Session on Emerging Techniques and Applications in Multi-Objective Reinforcement Learning at the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning 2015*, pages 53–58.

Kuleshov, V. and Precup, D. (2014). Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*.

Kurniawati, H., Hsu, D., and Lee, W. (2008). SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems*.

Liu, Q. and Ihler, A. (2013). Variational algorithms for marginal MAP. *Journal of Machine Learning Research*, 14:3165–3200.

Liu, Q. and Ihler, A. T. (2011). Bounding the partition function using Hölder's inequality. In *ICML 2011: Proceedings of the 28th International Conference on Machine Learning*, pages 849–856.

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

MacDermed, L. C. and Isbell, C. (2013). Point based value iteration with optimal belief compression for Dec-POMDPs. In *NIPS 2013: Advances in Neural Information Processing Systems 26*, pages 100–108.

Madani, O., Hanks, S., and Condon, A. (2003). On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1):5–34.

Manne, A. S. (1960). Linear programming and sequential decisions. *Management Science*, 6(3):259–267.

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

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

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

Marinescu, R., Razak, A., and Wilson, N. (2012). Multi-objective influence diagrams. In *UAI 2012: Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pages 574–583.

Mateescu, R. and 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*, pages 380–387.

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

Moffaert, K. V. and Nowé, A. (2014). Multi-objective reinforcement learning using sets of Pareto dominating policies. *Journal of Machine Learning Research*, 15:3483–3512.

Monahan, G. (1982). State of the art — a survey of partially observable Markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16.

Monostori, L., Váncza, J., and Kumara, S. R. (2006). Agent-based systems for manufacturing. *CIRP Annals-Manufacturing Technology*, 55(2):697–720.

Nair, R., Varakantham, P., Tambe, M., and Yokoo, M. (2005). Networked distributed POMDPs: A synthesis of distributed constraint optimization and POMDPs. In *AAAI 2005: Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 133–139.

Nayyar, A., Mahajan, A., and Teneketzis, D. (2013). Decentralized stochastic control with partial history sharing: A common information approach. *IEEE Transactions on Automatic Control*, 58:1644–1658.

Oliehoek, F. A. (2010). *Value-based planning for teams of agents in stochastic partially observable environments*. PhD thesis, University of Amsterdam.

Oliehoek, F. A. and Amato, C. (2014). Dec-POMDPs as non-observable MDPs. IAS technical report IAS-UVA-14-01, Amsterdam, The Netherlands.

Oliehoek, F. A., Spaan, M. T. J., and Witwicki, S. (2015). Factored upper bounds for multiagent planning problems under uncertainty with non-factored value functions. pages 1645–1651.

Oliehoek, F. A., Whiteson, S., and Spaan, M. T. J. (2012). Exploiting structure in cooperative Bayesian games. In *UAI 2012: Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pages 654–664.

Ong, S. C. W., Png, S. W., Hsu, D., and Lee, W. S. (2010). Planning under uncertainty for robotic tasks with mixed observability. *The International Journal of Robotics Research*, 29(8):1053–1068.

Pardoe, D. M. (2011). *Adaptive trading agent strategies using market experience*. PhD thesis, University of Texas at Austin.

Pearl, J. (1988). *Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Perny, P., Weng, P., Goldsmith, J., and Hanna, J. P. (2013). Approximation of Lorenz-optimal solutions in multiobjective Markov decision processes. In *UAI 2013: Proceedings of the Twenty-Ninth Conference on Uncertainty In Artificial Intelligence*, pages 508–517.

Pineau, J., Gordon, G., and Thrun, S. (2006). Anytime point-based approximations for large POMDPs. *Journal of Artificial Intelligence Research*, 27:335–380.

Poupart, P., Kim, K., and Kim, D. (2011). Closing the gap: Improved bounds on optimal POMDP solutions. In *ICAPS 2011: Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, pages 194–201.

Puterman, M. L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

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

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

Roijers, D. M., Whiteson, S., Ihler, A. T., and Oliehoek, F. A. (2015a). Variational multi-objective coordination. In *MALIC 2015: NIPS Workshop on Learning, Inference and Control of Multi-Agent Systems*.

Roijers, D. M., Whiteson, S., and 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*, pages 309–323.

Roijers, D. M., Whiteson, S., and Oliehoek, F. A. (2013c). Multi-objective variable elimination for collaborative graphical games. In *AAMAS 2013: Proceedings of the Twelfth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1209–1210. Extended Abstract.

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

Roijers, D. M., Whiteson, S., and Oliehoek, F. A. (2015b). Computing convex coverage sets for faster multi-objective coordination. *Journal of Artificial Intelligence Research*, 52:399–443.

Roijers, D. M., Whiteson, S., and Oliehoek, F. A. (2015c). Point-based planning for multi-objective POMDPs. In *IJCAI 2015: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 1666–1672.

Roijers, D. M., Whiteson, S., Vamplew, P., and Dazeley, R. (2015d). Why multi-objective reinforcement learning? In *EWRL 2015: Proceedings of the 12th European Workshop on Reinforcement Learning*.

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

Rollón, E. and 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*, pages 98–105. ACM.

Russell, S., Norvig, P., and Intelligence, A. (1995). *Artificial Intelligence: A modern approach*. Prentice-Hall, Egnlewood Cliffs.

Scharpff, J., Roijers, D. M., Oliehoek, F. A., Spaan, M. T., and de Weerdt, M. M. (2015). Solving multi-agent MDPs optimally with conditional return graphs. In *MSDM 2015: Proceedings of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains*.

Scharpff, J., Roijers, D. M., Oliehoek, F. A., Spaan, M. T., and de Weerdt, M. M. (2016). Solving transition-independent multi-agent MDPs with sparse interactions. In *AAAI 2016: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. To Appear.

Scharpff, J., Spaan, M. T. J., de Weerdt, M. M., and Volker, L. (2013). Planning under uncertainty for coordinating infrastructural maintenance. In *Proceedings of*

the International Conference on Automated Planning and Scheduling, pages 425–433.

Sen, A. K. (1995). *Collective choice and social welfare*. Elsevier, Amsterdam, the Netherlands. Fourth impression.

Shani, G., Pineau, J., and Kaplow, R. (2013). A survey of point-based POMDP solvers. *Journal of Autonomous Agents and Multi-Agent Systems*, 27(1):1–51.

Soh, H. and Demiris, Y. (2011a). Evolving policies for multi-reward partially observable Markov decision processes (MR-POMDPs). In *GECCO'11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 713–720.

Soh, H. and Demiris, Y. (2011b). Multi-reward policies for medical applications: Anthrax attacks and smart wheelchairs. In *GECCO'11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 471–478.

Sondik, E. J. (1971). *The optimal control of partially observable Markov decision processes*. PhD thesis, Stanford University.

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

Spaan, M., Amato, C., and Zilberstein, S. (2011). Decision making in multiagent settings: Team decision making. Tutorial slides.

Spaan, M. T. (2012). Partially observable Markov decision processes. In Wiering, M. A. and Van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 387–414. Springer Berlin/Heidelberg.

Spaan, M. T. J. and Vlassis, N. A. (2011). Perseus: Randomized point-based value iteration for pomdps. *CoRR*, abs/1109.2145.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.

Szita, I. (2012). Reinforcement learning in games. In Wiering, M. A. and Van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 539–577. Springer Berlin/Heidelberg.

Tambe, M. (2011). *Security and game theory: Algorithms, deployed systems, lessons learned*. Cambridge University Press.

Tesauro, G., Das, R., Chan, H., Kephart, J. O., Lefurgy, C., Levine, D. W., and Rawson, F. (2007). Managing power consumption and performance of computing systems using reinforcement learning. In *NIPS 2007: Advances in Neural Information Processing Systems 20*, pages 1497–1504.

Thiébaux, S., Gretton, C., Slaney, J. K., Price, D., Kabanza, F., et al. (2006). Decision-theoretic planning with non-markovian rewards. *Journal of Artificial Intelligence Research*, 25:17–74.

Vamplew, P., Dazeley, R., Barker, E., and Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In *AI 2009: Proceedings of the Twenty-Second Australasian Joint Conference on Artificial Intelligence*, pages 340–349.

Van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. In Wiering, M. A. and Van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 207–251. Springer Berlin/Heidelberg.

Van Moergestel, L. J. (2014). *Agent Technology in Agile Multiparallel Manufacturing and Product Support*. PhD thesis, Utrecht University.

Van Moffaert, K., Brys, T., Chandra, A., Esterle, L., Lewis, P. R., and Nowé, A. (2014). A novel adaptive weight selection algorithm for multi-objective multi-agent reinforcement learning. In *IJCNN 2014: Proceedings of the 2013 International Joint Conference on Neural Networks*, pages 2306–2314.

Vlassis, N., Elhorst, R., and Kok, J. (2004). Anytime algorithms for multiagent decision making using coordination graphs. In *IEEE SMC 2004: Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 953–957.

Vlassis, N., Ghavamzadeh, M., Mannor, S., and Poupart, P. (2012). Bayesian reinforcement learning. In Wiering, M. A. and Van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 359–386. Springer Berlin/Heidelberg.

Wainwright, M. J. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2):1–305.

White, C. C. and Kim, K. M. (1980). Solution procedures for solving vector criterion Markov decision processes. *Large Scale Systems*, 1:129–140.

White, D. (1982). Multi-objective infinite-horizon discounted Markov decision processes. *Journal of Mathematical Analysis and Applications*, 89(2):639 – 647.

Whiteson, S. and Roijers, D. M. (2015). Multi-objective decision making. In *IJCAI 2015 tutorials*.

Wiering, M. A. and Van Otterlo, M. (2012). Reinforcement learning: State-of-the-art. In *Adaptation, Learning, and Optimization*, volume 12. Springer.

Wiering, M. A., Withagen, M., and Drugan, M. M. (2014). Model-based multi-objective reinforcement learning. In *ADPRL 2014: Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 1–6.

Wiggers, A. J., Oliehoek, F. A., and Roijers, D. M. (2015). Structure in the value function of zero-sum games of incomplete information. In *MSDM 2015: Proceedings of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains*.

Wilson, N., Razak, A., and Marinescu, R. (2015). Computing possibly optimal solutions for multi-objective constraint optimisation with tradeoffs. In *IJCAI 2015: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 815–821.

Witwicki, S. J. and Durfee, E. H. (2010). Influence-based policy abstraction for weakly-coupled Dec-POMDPs. In *ICAPS 2010: Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 185–192.

Wray, K. H. and Zilberstein, S. (2015). Multi-objective POMDPs with lexicographic reward preferences. In *IJCAI 2015: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 1719–1725.

Yahyaa, S. Q., Drugan, M. M., and Manderick, B. (2014). The scalarized multi-objective multi-armed bandit problem: an empirical study of its exploration vs. exploitation tradeoff. In *IJCNN 2014: Proceedings of the 2014 International Joint Conference on Neural Networks*, pages 2290–2297.

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

Zintgraf, L. M., Kanters, T. V., Roijers, D. M., Oliehoek, F. A., and Beau, P. (2015). Quality assessment of MORL algorithms: A utility-based approach. In *Benelearn 2015: Proceedings of the Twenty-Fourth Belgian-Dutch Conference on Machine Learning*.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and Da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.

Abstract

Decision making is hard. It often requires reasoning about uncertain environments, partial observability and action spaces that are too large to enumerate. In such complex decision-making tasks decision-theoretic agents, that can reason about their environments on the basis of mathematical models and produce *policies* that optimize the utility for their users, can often assist us.

In most research on decision-theoretic agents, the desirability of actions and their effects is codified in a scalar reward function. However, many real-world decision problems have multiple objectives. In such cases the problem is more naturally expressed using a vector-valued reward function. Rather than having a single optimal policy, we then want to produce a set of policies that covers all possible preferences between the objectives. We call such a set a *coverage set*.

In this dissertation, we focus on decision-theoretic planning algorithms that produce the *convex coverage set (CCS)*, which is the optimal solution set when either: 1) the user utility can be expressed as a weighted sum over the values for each objective; or 2) policies can be stochastic.

We propose new methods based on two popular approaches to creating planning algorithms that produce an (approximate) CCS by building on an existing single-objective algorithm. In the *inner loop* approach, we replace the summations and maximizations in the inner most loops of the single-objective algorithm by cross-sums and pruning operations. In the *outer loop* approach, we solve a multi-objective problem as a series of scalarized problems by employing the single-objective method as a subroutine.

Our most important contribution is an outer loop framework that we call *optimistic linear support (OLS)*. As an outer loop method OLS builds the CCS incrementally. We show that, contrary to existing outer loop methods, each intermediate result is a bounded approximation of the CCS with known bounds (even when the single-objective method used is a bounded approximate method as well) and is guaranteed to terminate in a finite number of iterations.

We apply OLS-based algorithms to a variety of multi-objective decision problems,

and show that it is more memory-efficient, and faster than corresponding inner loop algorithms for moderate numbers of objectives. We show that exchanging subroutines in OLS is relatively easy and illustrate the importance on a complex planning problem. Finally, we show that it is often possible to reuse parts of the policies and values, found in earlier iterations of OLS, to hot-start later iterations of OLS. Using this last insight, we propose the first method for multi-objective POMDPs that employs point-based planning and can produce an ε-CCS in reasonable time.

Overall, the methods we propose bring us closer to truly practical multi-objective decision-theoretic planning.

Samenvatting

Beslissingen nemen is moeilijk. Het is vaak nodig om te redeneren over onzekerheid in de omgevingsdynamiek, partiële observeerbaarheid en actieruimtes die te groot zijn om te enumereren. In zulke complexe beslistaken kunnen beslistheoretische agents, die aan de hand van mathematische modellen over omgevingen kunnen redeneren en *policies* kunnen formuleren die het nut voor de gebruiker optimaliseren, ons helpen.

In het meeste onderzoek over beslistheoretische agents wordt de wenselijkheid van acties en de gevolgen daarvan geëncodeerd in een scalaire beloningsfunctie. Echter, veel realistische beslisproblemen hebben meerdere doelen tegelijk, waardoor het natuurlijker is om vectoren als beloning te gebruiken. In plaats van een enkele optimale policy, willen wij in dit geval een verzameling van policies genereren die alle mogelijke voorkeuren met betrekking tot de doelen dekt. Wij noemen dit een *coverage set*.

In dit proefschrift, focussen wij op belistheoretische planalgoritmes die een *convex coverage set (CCS)* produceren. De CCS is de optimale oplossing als: 1) het nut voor de gebruiker uitgedrukt kan worden als een gewogen som over de verschillende doelen; of 2) stochastische policies zijn toegestaan.

Wij stellen nieuwe methodes voor met behulp van twee methodes voor het creëren van CCS-planalgoritmes op basis van bestaande enkeldoelige algoritmes. Bij *binnenloopaanpak* vervangen wij de sommaties en maximisaties in de binnenste loops van het algoritme door cross sums en pruningoperaties. In de *buitenloopaanpak*, lossen we een meerdoelig beslisprobleem op als een serie gescalariseerde problemen, waarbij we het enkeldoelige algoritme als subroutine gebruiken.

Onze belangrijkste bijdrage is een buitenloop-framework dat we *optimistic linear support (OLS)* noemen. Als buitenloopmethode bouwt OLS de CCS iteratief op. We laten zien dat, in tegenstelling tot bestaande buitenloopmethodes, op elk tussenresultaat van OLS een benaderingsgarantie gegeven kan worden (zelfs als de enkeldoelige subroutine zelf een benaderingsalgoritme is), en dat OLS binnen een eindig aantal iteraties termineert.

We passen op OLS gebaseerde algoritmes toe op verschillende meerdoelige beslisproblemen en tonen aan dat OLS zuiniger met geheugen is, en sneller is dan overeenkom-

stige binnenloopalgoritmes voor gematigte hoeveelheden doelen. We laten zien dat het vervangen van subroutines binnen OLS relatief eenvoudig is, en tonen het belang daarvan aan met een complex planprobleem. Als laatste laten we zien dat het vaak mogelijk is om delen van in eerdere iteraties gevonden policies en waarden daarvan te hergebruiken in latere iteraties, om de enkeldoelige subroutines een vliegende start te geven. Gebruik makend van dit laatste inzicht, stellen wij de eerste planmethode voor meerdoelige POMDPs voor die gebruik maakt van point-based planalgoritmes en een ε-CCS kan produceren binnen redelijke tijd.

De methodes die wij in dit proefschrift voorstellen brengen ons dichter bij in de praktijk makkelijk toepasbare meerdoelige beslistheoretische planalgoritmes.

Overview of Publications

Conforming to the regulations of the University of Amsterdam, we provide an overview of the publications of the author (Diederik M. Roijers) that were used as the basis for this dissertation. In accordance with the regulations, we mention the contributions of the author of this dissertation on these publications.

We note that it is difficult to attribute the ideas to a single author as we did a lot of brainstorming and iterated many times to improve our initial ideas, for all our papers and articles.

Core Publications

These are the papers on which this disseration is based:

- Roijers et al. (2013a) — In this article, we (Diederik M. Roijers, Peter Vamplew, Shimon Whiteson and Richard Dazeley) survey the field of decision-theoretic planning and learning in multi-objective Markov decision processes. We use part of the literature survey in Section 5.1.1 of this dissertation. While writing the survey article, Diederik Roijers and Shimon Whiteson focussed on the utility-based approach, the scenario's and the taxonomy (that we use in Chapters 1 and 2 in this dissertation), as well as some of the future work (including the ESR formulation, that we use in Section 6.2.2). Most of the initial ideas for these topics were first conceived by the Diederik Roijers. Note however, that we did a lot of brainstorming and iterated many times to improve our initial ideas. Peter Vamplew and Richard Dazeley focussed on the literature overview and the applications. All authors provided feedback on all sections of the article.

- Roijers et al. (2013b) and Roijers et al. (2013c) — In these two papers we (Diederik M. Roijers, Shimon Whiteson and Frans A. Oliehoek) first propose the CMOVE algorithm (Section 4.3.1). Although the idea of CMOVE was already conceived in the project proposal, before Diederik Roijers was hired as the main researcher on this topic, he contributed some of the core elements of the

CMOVE algorithm and the paper, such as the tagging scheme, the analysis in terms of the size of the intermediate coverage sets, the implementation, and the experiments.

- Roijers et al. (2014b) — In this paper, we (Diederik M. Roijers, Shimon Whiteson and Frans Oliehoek) first proposed OLS (Chapter 3), in the context of MO-CoGs (Chapter 4). Diederik Roijers was the primary researcher; the other authors provided support and feedback.

- Roijers et al. (2014a) — In this paper, we (Diederik M. Roijers, Joris Scharpff, Matthijs T.J. Spaan, Frans A. Oliehoek, Mathijs de Weerdt and Shimon Whiteson) extend OLS to be able to handle bounded approximate single-objective subroutine (as we describe in Section 3.5), as well as a different outer loop method that we do not treat in this dissertation. The (initial version of the) correctness proof for approximate OLS was constructed in a brainstorm session with Diederik Roijers, Joris Scharpff and Mathijs de Weerdt. In this paper, the work was divided more or less evenly between the first two authors, Diederik Roijers focussing more on the algorithmics, and Joris Scharpff more on the implementation and experiments; the other authors provided support and feedback.

- Roijers et al. (2015b) — In this article, we (Diederik M. Roijers, Shimon Whiteson and Frans Oliehoek) collect our findings for MO-CoGs (Chapter 4) from our earlier work (i.e., CMOVE and VELS), and propose the memory-efficient algorithms CTS and TSLS. Diederik Roijers was the primary researcher; the other authors provided support and feedback.

- Roijers et al. (2015c) —In this paper, we (Diederik M. Roijers, Shimon Whiteson and Frans Oliehoek) propose the OLSAR algorithm (Section 5.3), and the general idea for reuse in OLS (Section 3.6). Diederik Roijers was the primary researcher; the other authors provided support and feedback.

- Roijers et al. (2015a) — In this paper, we (Diederik M. Roijers, Shimon Whiteson, Alex Ihler and Frans Oliehoek) propose the VOLS algorithm (Section 4.4.5). Diederik Roijers was the primary researcher; the other authors provided support and feedback. Alex Ihler provided the single-objective solver code (WMB).

Other Papers

There are two more papers that have been (partially) used in this dissertation:

- Zintgraf et al. (2015) — In this paper, we (Luisa M. Zintgraf, Timon V. Kanters, Diederik M. Roijers, Frans A. Oliehoek and Philipp Beau) use the utility-based approach to prove a bound on the utility loss of ε-PCSs, and propose a generalized MOMDP benchmark problem. The results of the proof were included in

Section 2.1.3. Luisa Zintgraf focussed on the proof. Timon Kanters and Philipp Beau focussed on the benchmark. Diederik Roijers and Frans Oliehoek provided feedback and were involved in brainstorm sessions; this work was done in the context of a student project supervised by Diederik Roijers and Frans Oliehoek.

- Scharpff et al. (2016) — In this paper, we (Joris Scharpff, Diederik M. Roijers, Frans A. Oliehoek, Matthijs T.J. Spaan, and Mathijs M. de Weerdt) propose the CoRe algorithm, which we use as a single-objective subroutine inside OLS in Section 5.2. Joris Scharpff had the lead on this paper. Joris Scharpff and Diederik Roijers did most of the research. Diederik Roijers' focus was mainly on the algorithmics. The other authors provided support and feedback.