Collaborative Multi-Agent Planning with Partial Observability and Interacting Actions

**Thesis submitted in partial fulfillment
of the requirements for the degree of
"DOCTOR OF PHILOSOPHY"**

by

Shashank Shekhar

**Submitted to the Senate of Ben-Gurion University
of the Negev**

March 2021

Beer-Sheva

Collaborative Multi-Agent Planning with Partial Observability and Interacting Actions

Thesis submitted in partial fulfillment
of the requirements for the degree of
"DOCTOR OF PHILOSOPHY"

by

Shashank Shekhar

Submitted to the Senate of Ben-Gurion University
of the Negev

Approved by the advisor

Approved by the Dean of the Kreitman School of Advanced Graduate Studies

March 2021

Beer-Sheva

This work was carried out under the supervision of

Professor Ronen I. Brafman

In the Department of Computer Science

Faculty of Natural Sciences

Research-Student's Affidavit when Submitting the Doctoral Thesis for Judgment

I, Shashank Shekhar, whose signature appears below, hereby declare that:

___ I have written this Thesis by myself, except for the help and guidance offered by my Thesis Advisors.

___ The scientific materials included in this Thesis are products of my own research, culled from the period during which I was a research student.

Date: **14 March 2021** Student's name: **Shashank Shekhar** Signature: Shashank Shekhar

This thesis is dedicated to my beloved parents for always supporting me and relentlessly encouraging me to strive for excellence.

"Happiness is when what you think, what you say, and what you do are in harmony."

– *Mahatma Gandhi*

Acknowledgements

I would like to thank my doctoral advisor, Prof. Ronen I. Brafman, for his valuable guidance through each stage of the process. Because of his spot-on research advice over the years, this whole doctoral period became an incredible period of my life. I enjoyed my time working with him and felt very fortunate for that. His constant encouragement to explore multiple research paths, some of which were only distantly related to what we have worked on together, allowed me to mature as a researcher, making the most out of my doctoral studies.

Our research collaborator, Dr. Guy Shani, was instrumental, too, in defining the path of my research. I am grateful to him for advising me like a second advisor. His insights on multi-agent planning under uncertainty and partial observability helped to develop my thoughts much better on this topic. Apart from that, I must thank him for all the help he provided me with in sorting technical issues out; throughout these years.

I would also like to thank Dr. Roni Stern. Although my joint work with him is not part of this thesis, working with him and his group has broadened my view of the field of multi-agent planning & pathfinding (MAPF) and heuristic search, improving my research skills immensely. Their motivation, knowledge, and hard work truly inspire me.

A big thank goes to all my friends in Ben-Gurion University of the Negev for their availability. They were always there for me, irrespective of my good times or bad.

A special thank goes to Beer-Sheva Cricket Club, Beer-Sheva, for having me among those great club members. Congratulations to them on having such a great cricket club in Israel. I enjoyed most of my Saturdays, which I spent with these guys. Playing with them was always highly refreshing and helped me feel that I am still in my hometown.

Finally, my deepest, never-ending gratitude goes to my parents and the family. Their hard work, sacrifices, and most importantly, their unconditional support to my “do whatever makes you happy” attitude are great inspirations. I would also like to thank my extended family for being highly supportive throughout these years.

Abstract

Collaborative Multi-Agent Planning (MAP) under uncertainty with partial observability is a notoriously difficult problem. Such MAP problems are often modeled as Dec-POMDPs, or its qualitative variant, QDec-POMDPs, which is essentially a MAP version of contingent planning. The QDec-POMDP model was introduced with the hope that its simpler, non-probabilistic structure will allow for better scalability. Indeed, at least with deterministic actions, the recent Iterative MAP algorithm (IMAP) scales much better than comparable Dec-POMDP algorithms.

In this work, we describe two new approaches for solving Deterministic QDec-POMDPs, which share a common factored framework that is motivated by the main ideas behind factored algorithms for classical planning, in that a problem factoring is achieved by defining the planning problem as the MAP problem.

In our first approach, we start by finding a solution to a MAP problem where the results of observation are available to all agents. This is essentially a single-agent planning problem for the entire team, called the *team-problem*. Then, we project its solution tree into sub-trees, one per agent, and let each agent transform its projected tree into a local tree executable online. If all agents succeed, we combine the trees into a valid joint-plan. Otherwise, we continue to explore the space of team solutions. We call it the *QDec-FP* approach (stands for Qualitative Decentralized Partially Observable Markov Decision Processes using Factored Planning). In the second approach, we describe a planner that uses richer information about agents' knowledge to improve the team planning process to generate more "informed" single-agent plans for the entire team. Modeling individual agent's knowledge also supports modeling communication between agents and planning using it. In particular, we discuss the idea of *Signaling*, where agents share knowledge by changing the state of the world, and we named this algorithm *QDec-FPS*, which stand for QDec-FP that supports *S*ignaling. We also discuss the soundness, completeness, and other theoretical properties of the two approaches.

To test the properties of our approaches, we model and describe a number of new MAP domains, and we empirically show that the QDec-FP planner performs and scales much better than the IMAP planner on both old and new MAP domains. We note that previously, IMAP was the best QDec-POMDP solver, scaling much better than recent Dec-POMDP planners.

Consequently, QDec-FP scales to even larger problems, much beyond that attainable by state-of-the-art Dec-POMDP solvers. Then, we test the scalability of QDec-FPS by comparing it with QDec-FP and show that QDec-FPS performance is overall much better and has better applicability. Moreover, we show that this new approach solves MAP problems that cannot be practically solved by QDec-FP, as they require the use of signaling.

One core property of the MAP domains used above for evaluating the QDec-POMDP frameworks is that they often model loosely-coupled agents. These MA domains assume that the single-agent actions do not interact with each other when performed concurrently. Or, that if two or more single-agent actions interact, this is modeled using an explicit, *collaborative action* that comprises these interacting actions. In some sense, such multi-agent planning domains model useful concurrency if required to achieve *useful* things but do not care about efficient planning.

To better investigate efficient description of domains in which actions interact more strongly, the next part of this work studies *interacting actions* – actions whose joint-effect differs from the union of their individual effects. These actions are challenging both to represent and plan with due to their combinatorial nature. We study the representation of these actions in the simpler MAP settings with full observability and with no uncertainty. So far, there have been few attempts to provide a succinct language for representing them that can also support *efficient* centralized planning and distributed privacy-preserving planning. We suggest an approach for representing interacting actions succinctly and show how such a domain model can be compiled into a standard single-agent planning problem as well as privacy-preserving multi-agent planning. We test the performance of our method on several novel domains involving interacting actions and privacy.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Over the last twenty years, our world has become much more "interconnected". Even if we see things historically, this interconnectedness has been part of our life. This implies that there are multiple types of interactions possible between different aspects of our lives, especially when we see each human being, an independent "entity". Interactions can include people fighting wars together, engaging in politics, playing games like chess, and several other options. Realization of the central role of multi-agent interaction has led to the formalization of Game Theory [101]. Over sixty years ago, (computer) scientists developed this idea of a real "entity" (or "agent") into the concepts of Artificial Intelligence (AI). They thought of an artificial entity acting in an environment while interacting with it simultaneously over a sequence of (time-)steps. Moreover, they also considered multiple entities interacting with each other in some sense, passing information to each other, assisting each other to achieve "joint-goals" collaboratively, and more. Later, it was realized that even while working as a team, agents may want to *secure* some information that they do not wish to share with other team members, exactly as we humans do. Even in a cooperative setup in which entities have a joint utility, they remain distinct entities and often have privacy concerns. Thus, in principle, we reach a view of multiple entities working in an environment, interacting with each other and the environment, while, if needed, they aspire to keep their secrets unrevealed [27, 36, 95]. These entities are "intelligent", in the sense that they have computational capabilities and multiple smart sensors onboard, which they use to make decisions for acting in the environment.

Today, we observe the interconnectedness of multiple entities like routers, robots, computers and people in our surroundings, whether we speak of our physical surroundings or a virtual surrounding. This gives rise to abundant examples of real and virtual multi-agent systems around us. With the increasing penetration of the Internet of Things (IoT) and advances in robotic technologies, in the future, their number will increase. In many of these systems, the entities – be they smartphones, autonomous cars, intelligent robots, smart home-appliances (part of IoT) – work together to achieve joint goals. We expect them to solve a variety of real-world problems in the future, and to cooperate and coordinate with each other to achieve our goals.

With the increasing number and sophistication of current agents/devices, it is critical to properly control their behavior, i.e., so that they act, communicate, and coordinate effectively. This task becomes much more challenging when the environment is partially observable and unpredictable and when the entities have noisy sensors. The reason for this is that each entity can typically sense only its immediate surroundings. Because the entities often differ in their location and capabilities, each entity learns a different aspect of the environment. This situation leads to entities with partial and different knowledge of the state of the world. Therefore, making sure that their overall, combined behavior is effective is a non-trivial task. We note that from now onwards we will use the terms "entity" and "agent" interchangeably.

Given the complexity of the problem alluded to above, it is clear that manually constructing good controllers for operating a multi-agent system with even a small number of entities in a partially observable and uncertain environment is a non-trivial task. Hence, the area of automated multi-agent planning seeks to automatically generate such controllers given a suitable specification of the domain and the problem. The goal of this thesis is to address the problem of planning for a cooperative multi-agent system. Such cooperative systems form a major part of the multi-agent systems alluded to earlier. Our primary focus is on collaborative multi-agent planning with partial observability and collaborative multi-agent planning with interacting actions and privacy. We briefly describe our contributions in later sections of this chapter and more thoroughly in the upcoming chapters.

But first, let us review the idea of automated planning. Ghallab, Nau, and Traverso (2004) define planning [13] as follows: *Planning is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. This deliberation aims at achieving as best as possible some prestated objectives.* Planning is traditionally considered a centralized process, in which a "single entity" participates in building a plan that achieves defined objectives of a planning task.

Classical planning is the simplest form of planning, and models a centralized process in which a "planner" plans in a fully observable environment and assumes that it also controls this environment. Moreover, actions have deterministic outcomes, and the planner always starts from a fully known initial state. But even in its simplest form, planning is known to be PSPACE hard [21]. Therefore, planning researchers often aim at understanding the nature of planning problems [2, 14]. They exploit their understanding for designing new algorithms that expand our horizon of practically solvable problems. Most work in this area has focused on devising domain-independent approaches, i.e., approaches that can be applied to solve any planning problems whose description contains a domain model. Such a model comprises a finite set of variables (*Boolean* or *multi-valued*) and actions, and each action is described by its preconditions and effects. In general, much work in automated planning is centered on devising domain-independent technology [33] in a variety of real-world applications like space-exploration, military operations, routing, rescue missions, logistics, and others.

Multi-Agent Planning (MAP) is the problem of planning in the presence of multiple entities who plan and act together in a shared environment [60]. Some real-world examples that can be viewed as MAP problems are multi-agent logistics problems and multi-agent search and rescue operations, etc. The presence of several entities, and therefore the requirement to keep in mind the number of different actions they could perform and how their goals are aligned with each other, makes such problems more complex than the classical planning problem. Assuming, *e.g.*, that the agents are cooperative, can make the MAP problem somewhat simpler.

Agents are known as *non-strategic* or *cooperative* agents if they do not have any private objectives to fulfill and work in a cooperative manner towards solving a planning problem with a common goal. Agents are known as *competitive* or *self-interested* or *strategic* if they participate in planning with an “intention” to maximize their individual-profits, and are motivated towards accomplishing their own goals. Often, a framework (like in [58]) that models the MAP problem is, at a high level, an extension of the single-agent distributed planning (or problem-solving) framework [14, 102]. Once the framework is formalized, we use this very framework for planning and decision-making purposes for each agent in the team.

One may ask why one cannot simply solve the MAP problem as a single-agent problem in which the agents are simple entities, like other objects. The single-agent solver would solve the MAP problem on a centralized system. Later, a post-processing step would split the obtained centralized solution to generate an individual plan for each agent. However, this idea is not feasible for many systems. First and foremost, strategic agents – *i.e.*, agents that pursue their own interests – are unlikely to agree to share their information and accept a policy from a third party (although secure computation schemes can make this possible in some scenarios). For example, taxi drivers might not share information that breaches their privacy or affects their profits. Similarly, cooperative agents with privacy concerns are not likely to agree to such a centralized approach. But more fundamentally, the planning process must take account of the potentially multiple, different states of information of the agents at the execution time. These can be due to different initial information states, and in distributed systems with partial observability, naturally arise due to the fact that different agents receive different signals from the environment. This situation is quite different from that of single-agent planning, in which we must reason about a single belief state only.

But while naively solving MAP centrally as if it is a single-agent planning problem is often not possible, planning independently for each agent, while having potential advantages, is also often problematic because the agents' actions might be interacting [82]. Hence, independent planning can produce plans that conflict during execution. In these circumstances, we need centralized planning to produce a complete, non-conflicting executable plan for the agents that takes the nature and structure of the MAP problem into account.

Much work has been done within the general topic of MAP. Following Durfee (2001), here are a few possible options for automated multi-agent planning for multiple agents [26]. First, there is *distributed* planning for *distributed* plans, where both the planning process and the obtained resulting plans for the agents are distributed. In that case, an entity does not need to know the complete problem specification, and entities could have only partial information about the environment. Second, there is *distributed* planning for *centralized* plans where, to solve a single-agent problem, we use multiple computational devices, which helps parallelize the planning process and makes the overall process faster. Later, the generated individual solution plans are combined to form one complete solution plan for the original problem. Third, there is *centralized* planning for *distributed* plans, where a centralized solver solves a MA problem such that the obtained solution is distributed among the acting agents.

This third option is often the most natural to follow, as it ensures suitable coordination between the plans executed by different agents. Such coordination may call for maintaining some ordering constraints among the actions of different agents, and in particular, executing certain interacting actions concurrently to achieve their desired effect. Indeed, because the main reason for centralized planning is to enforce proper coordination, it may suffice to achieve such coordination by solving an abstract (relaxed) version of the MAP problem centrally in order to generate a skeleton that guides agent-level planning, making the overall planning process much easier and more efficient. Here, centralized planning is used only for coordinating the agents' action interaction points. Based on these points, some commitments are generated for each agent, and these commitments need to be met and, hence, they carefully direct the agent while it plans for itself.

The thesis primarily focuses on this third option, *centralized* planning for *distributed* online execution, which we believe to be the best strategy among the three for the class of cooperative multi-agent problems that this work targets. We focus on problems in which agents have only partial information about the state of the world. However, towards the end, while dealing with agents' privacy when the actions of the agents are interacting, we also discuss privacy-preserving planning [13, 53, 58, 80, 96] with interacting actions [12, 83].

The most widely used model for the class of cooperative MAP problems under partial observability we wish to deal with is the Decentralized POMDPs (Dec-POMDPs) model. Dec-POMDPs offer a rich stochastic model to capture MAP under uncertainty with partial observability [7, 8, 60, 79]. They are an extension, or rather, a generalization, of POMDPs for the case of multiple entities with possibly different states of information. Dec-POMDPs solutions consist of (different) policies for different agents. Dec-POMDP algorithms seek to generate such policies with the aim that when the agents execute their policies online, the expected cumulative discounted future reward for the team is maximized. The complexity of finding optimal policies for Dec-POMDPs is NEXP hard, making it difficult to solve instances of real life problems from this class.

A key requirement from each agent's policy in this framework is that the agent's choice of action only depends on its state of information [8]. An important realization is that part of one agent's policy could be to change the state of information of the another agent. This is essentially what communication does. But communication is just one explicit example of how agents can influence the state of information of other agents. A more general perspective is that agents can change the state of the world not only to further their goals, but also to affect the information state of another agent. For example, one agent may place the house key under the doormat to signal to a second agent that it is not at home. Later, the choice of action of the second agent may depend on this new information. Indeed, viewed this way, communication is simply an act of changing the state of some communication channel.

As an alternative to the Dec-POMDP framework, we study a conceptually simpler, non-stochastic framework for MAP under uncertainty with partial observability, called *Qualitative* Dec-POMDP (QDec-POMDP), which is a generalization of the single-agent contingent planning model [1, 43, 50, 55]. In contingent planning, a single-agent with sensing actions starts in an initially unknown state and seeks to achieve its goal regardless of its true initial state while using its sensing actions to learn useful information about its environment [1, 66, 68]. The solution to a contingent planning problem can be represented as a tree (or a graph). The non-leaf nodes of this tree represent actions, and the edges represent the observations made by the agent, while each leaf node denotes a state that satisfies the goal condition [43]. The contingent planning problem is the most general problem in the planning field, but is considered one of the hardest [74]. In QDec-POMDPs, we replace the quantitative probability distributions over possible states with qualitative sets of states. This model is less expressive since it specifies the possible outcome states without their likelihood. This can potentially help us solve qualitative versions of problems where it is difficult to obtain an accurate quantitative model, or when such a model is too complex to solve.

In terms of the worst-case complexity, *Qualitative* Dec-POMDPs are known to be no easier than Dec-POMDPs [18]. Nevertheless, a multi-agent contingent planning formalism offers two main advantages. First, it uses a propositional (*a.k.a.* factored) state model that is a much more convenient model to specify, and more succinct than flat state models that characterize much of the work on Dec-POMDPs. Second, much like contingent planning, QDec-POMDPs is more amenable to the use of current classical planning methods, which are quite powerful. This is especially true for deterministic QDec-POMDPs. Indeed, the authors of the QDec-POMDP model introduced this framework with the hope that, thanks to these advantages, it would scale up better. Initially, in Deterministic MAP problem with partial observability, QDec-POMDP algorithms scaled only somewhat better than contemporary Dec-POMDP algorithms. But more recent algorithms, like the Iterative MAP algorithm [5], scale much better than contemporary Dec-POMDP solvers. The majority of this thesis focuses on scaling to even larger deterministic QDec-POMDP models.

Following the solution structure of the contingent plan [43], the solution of QDec-POMDPs can be represented as a joint-policy tree (or a graph). The nodes of this tree are labeled by agents' joint-actions, while the edges are labeled by their joint observations. The leaf nodes of this joint-policy tree correspond to goal states. The high branching factor of such a joint-policy tree limits the scalability of this representation. However, one can obtain local policy trees, one for each agent, from this joint-policy tree, which has an exponentially smaller branching factor.

Figure 1.1 shows a joint-plan tree for a problem in the Box-Pushing domain [18]. It shows a joint-solution to a simple QDec-POMDP problem in the Box-Pushing domain. There is a grid-like structure having three cells marked as 1, 2, and 3. Two agents φ_1 and φ_2 start from cells 1 and 3, respectively. In each cell, there could be a box. The boxes in the left and right cells are light, and the one in the center is heavy. A light box needs only one agent to be pushed, while the heavy box requires two agents to push at the same time to move it out of the grid. Agents can sense whether a box is in a cell. The goal is to move all boxes out of the grid.

In many practical problems in cooperative settings, such as the multi-agent Logistics problem or the above Box-Pushing domain, the total number of action interactions is limited. Although some actions must be executed concurrently to succeed (e.g., as in the case of two agents pushing the heavy box), an agent can achieve parts of the joint goal through its own actions. Moreover, it can also assist other agents, making it possible for them to execute actions that serve the common objective. In such cases, we might want to try to solve as much of the problem locally, provided that the overhead of ensuring that the local policies are properly coordinated is not too large.

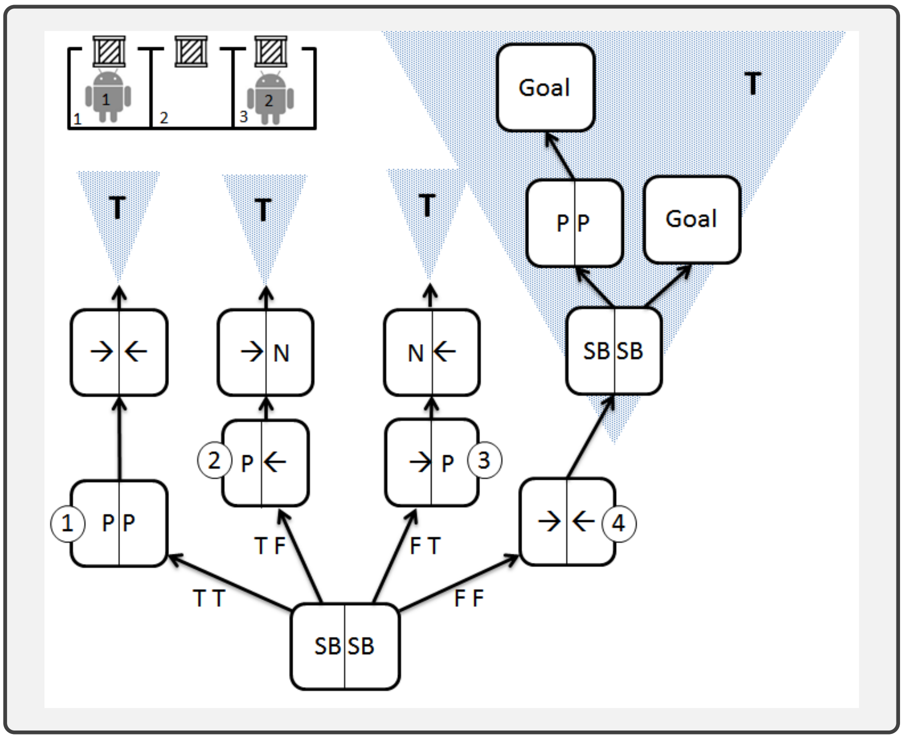

Fig. 1.1 This figure shows a joint-plan tree for the Box-Pushing domain with two agents and a possible joint policy tree with nodes labeled by joint actions. Possible agent actions are sensing a box at the current agent location (denoted *SB*), moving (denoted by arrows), pushing a box up (denoted as *P*), and *noop* (denoted as *N*). Note that the completely developed subtree (**T** shown by blue dashed vertical lines) in the rightmost branch is repeated in the other three branches. On the second level of the tree, nodes marked as 1 and 2 must have the same action for φ_1 (push up in this case), because φ_1 cannot distinguish between these two nodes. Likewise for nodes 2 and 4, with respect to φ_2 which cannot distinguish between them.

With this intuition in mind, in this thesis, we focus on solving *deterministic* QDec-POMDPs as well. In Chapter 3, we present an approach that solves multiple single-agent planning problems in order to solve this MA model. This approach is motivated by the main ideas behind factored algorithms for classical planning, in which problem factoring is accomplished by defining the problem as a MAP problem [14]. At a high-level, the approach works as follows. It starts by treating the entire MAP problem as a single-agent problem. We refer to this simplified problem as the *team problem*, and to its solution as the *team plan*. Our approach then decomposes the team solution into multiple parts, one for each agent. Each agent then fixes their part carefully. Finally, the solver centrally aligns the individual solutions to create an online executable joint policy tree.

The team planning process generalizes the agents' knowledge while relaxing the need to maintain different information states for different agents. Instead of that, it manages just one belief state for all the agents. Therefore, the process often leads to a team solution in which some agent cannot *fix* their projected part of the team solution. As a result, the planner must backtrack and start again with a new team problem.

To address this issue, we propose in Chapter 4 a modified algorithm for solving QDec-POMDPs, which makes team problems less abstract by reducing the overgeneralization they perform. The new approach models "agent-specific knowledge" at the team level, which helps generate informed team solutions. Modeling knowledge also helps model and plan the use of communication – either *implicit* or *explicit*. We use this ability to model implicit communication in this approach, where agents share information by changing the state of the world. That is, one agent can communicate certain information to another agent by acting in the world and changing its state in a pre-specified way, which eventually acts to change the other agent's knowledge about the world.

The empirical evaluation, covered in detail in Chapters 3 and 4, shows that both our approaches scale well beyond contemporary (*Qualitative*) Dec-POMDP solvers.

If one considers the nature and structure of the class of MAP problems modeled for evaluating our approaches, one can see that a "core" property appears in all of them. That is, that the agents modeled in these domains are loosely-coupled, and as a result, the agents' policies obtained are often loosely-coupled. We alluded to this property earlier. Moreover, we plan with these domains such that, at a given time instant, only a single action can be executed, either by one agent or by a set of collaborating agents. We refer to those latter actions involving multiple agents as *collaborative actions* [5, 82].

But what happens if we want to parallelize a sequential plan as much as possible, or if there is a rich set of interacting actions that we wish to describe, or if there are time, resource or other constraints that render the problem unsolvable unless actions are appropriately scheduled together. In such cases, we may need to try to schedule as many actions as possible concurrently. But how do we model the effect of diverse combinations of single agent actions?

In the worst case, we need to specify one collaborative action (better known in this case as *joint-action*) for every combination of actions executed by the agents. With enough agents, it is practically impossible to obtain an explicit specification due to its exponential size, let alone to plan with it efficiently. For that reason, when possible, we would like to have a succinct, implicit specification of the set of joint-actions, and to design algorithms that can effectively use such a specification.

To see the complications that can arise when specifying joint actions, consider what happens if a planning domain model contains a single-agent action, *lift(table)*, and a two-agent collaborative action *joint-lift(table).* In that case, the planner can concurrently schedule two single-agent *lift(table)* actions by two agents, which should be illegal as there exists an explicit two-agent *joint-lift(table)* action. Also, suppose that there exists a collaborative action, *joint-lift(table)*, comprising *three* agents. Now, the planner must be forbidden to apply three single-agent *lift(table)* actions together, or a two-agent *joint-lift(table)* and a single-agent *lift(table)* actions together. And what if an agent places an object on the table once other agents lift it?

To better investigate an efficient description of domains in which actions interact more strongly, the next part of this work studies *interacting actions* – those actions whose joint-effect differs from the union of their individual effects. Interacting actions are challenging both to represent and to plan with due to their combinatorial nature. To be succinct, a representation for joint-actions must be compositional. That is, there must be some way of deducing the effect of the concurrent execution of actions $\langle a_1, \ldots, a_n \rangle$, in which some components might be interacting, from the effects of smaller combinations.

Since representation and planning with interacting actions is non-trivial, most work on MAP algorithms ignores this issue and considers sequential actions or concurrent non-interacting actions. In the former case, joint-actions are not an issue, and sequential plans containing actions by different agents are generated. In the latter case, only actions that impact different variables or have the same effect on shared variables are considered. In that case, the effect of a joint-action is the union of effects of its component actions.

So far, there have been few attempts to provide a succinct language for representing these that can also support efficient centralized planning and distributed privacy-preserving planning. In Chapter 5, we suggest an approach for representing interacting actions succinctly and show how such a domain model can be compiled into a standard single-agent planning problem as well as into privacy-preserving multi-agent planning.

1.1 Main Contributions

This thesis describes two main contributions: the development of the state-of-the-art QDec-FP and QDec-FPS algorithms for solving QDec-POMDPs, and the development of a language for specifying interacting actions and a planning algorithm that can take this input as a representation, supporting a certain level of privacy as well.

Chapter 3 (based on [84]), proposes a factored planning approach to solve QDec-POMDPs. The approach works as follows. First, we solve a relaxed MAP problem in which we assume that communication is free and immediate. That means an observation made by one agent is available to all agents immediately. This is a *single-agent* planning problem and we call it a *team problem*, and its solution is called a *team solution*. From a team solution, we extract a sub-tree for each agent. We refer to this as the agent's *projection* of the policy tree. These agent-specific projected sub-trees are not likely to be executable. Thus, the next step in the algorithm is to let each agent *fix* its projected policy. If the agent succeeds, it generates a policy that the agent can execute online (provided the other agents execute their actions in the team policy). Of course, the single-agent problems may not be all *fixable*, in which case we must backtrack and seek a new team solution. But if they are all solvable, then we can get a sound joint policy tree for the original problem by taking the solutions of all the projected problems and properly aligning their actions. Overall, the factored approach performs much better than IMAP on the existing and new benchmark domains.

Chapter 4 (based on the work done in [86]), describes a planner that uses richer information about agents' knowledge to improve upon our current factored planning approach. While investigating our factored approach deeper, we realized that team planning problems exceedingly abstract the underlying problem. That is, many of the solutions of the team problem cannot be extended into real, distributed policies. The new algorithm uses enhanced reasoning about *individual* agents' knowledge in the team problem. Reasoning about individual agents' knowledge during team plan execution has two main advantages. First, it will lead to the generation of more informed team plans that are easier to extend to sound solutions, because the team planner adds an action only if the agent that executes it knows that its preconditions hold. This, in turn, forces the team planner to add additional, needed, sensing actions. Second, it allows us to model, within the team plan, the process of explicit and implicit communication, which we refer to as *signaling*. Note that to support signaling, the planner must model the knowledge of each agent within the team plan. Otherwise, the planner has no reason to insert signals into the plan, because signaling does not enhance the overall knowledge of the team. Our new approach is able to use signaling, and can solve problems that cannot be solved without it. Overall, this approach is shown to perform better than the original factored approach on the multi-agent planning problems with partial observability.

We also discuss the soundness and completeness properties of the two planners. In principle, both the planners are sound, but they are incomplete because they are implemented on top of an incomplete algorithm for single-agent contingent planning. However, one could ask whether they would be complete if the underlying single-agent solver is complete. Unfortunately, it turns out that in some cases the mechanism used in QDec-FPS to reason about agents' knowledge is too weak for this to be true. We explain and illustrate this problem in Section 4.4.3.

The last part of the thesis (Chapter 5) thoroughly studies efficient specifications of MA planning domains and planning using these domains, in which actions interact more strongly. Due to the combinatorial nature of interacting actions, they are non-trivial to represent and to plan with. There have been only a few attempts made to provide a succinct language for representing interacting actions, which can also support efficient centralized planning and distributed privacy-preserving planning. We suggest a method for representing interacting actions in a concise manner and show how a domain model, specifying strong action interactions, can be compiled into a standard single-agent planning problem and a privacy-preserving multi-agent planning problem. We evaluate the performance of our method on several novel domains involving interacting actions and privacy. This work was published in 2018 [82], and an extension of this work was published in 2020 [83].

1.2 Thesis Structure

Following this brief overview of the context, motivation, and main contributions of this thesis, the rest of this thesis is structured as follows:

- **Chapter 2:** This chapter provides a detailed background required to understand this thesis and an overview of related work.

- **Chapter 3:** This chapter describes an approach for solving deterministic QDec-POMDPs, based on factored planning [84]. We discuss its theoretical properties and empirically show that it scales much better than IMAP, and discuss the results.

- **Chapter 4:** This chapter describes an improvement over the factored planning approach presented in Chapter 3. First, we discuss the shortcomings of the previous method, which act as a motivation for the improved approach. Then, we explain how we can address them by introducing and explaining them via an example. We introduce *agent-specific knowledge modeling* at the level of team planning. Next, we demonstrate that knowledge modeling enables the approach to model and use communication between agents during team planning. We discuss its theoretical properties and show that it scales overall much better than the former method, and show that it has better applicability as well.

- **Chapter 5:** This chapter provides a thorough study of an approach for representing and planning with interacting actions in deterministic, fully-observable MAP problems [82, 83]. It puts forth an intuitive formalism for specifying joint-actions in a compositional way. To test our approach, we introduce a number of new domains and present the empirical results. This chapter also highlights and discusses subtle issues that arise when attempting to model and plan with interacting actions.

- **Chapter 6:** This chapter concludes with a summary of the main contributions, and presents some of the challenges for future work.

Chapter 2

Background and Related Work

Multi-agent planning exists at the intersection of automated planning and multi-agent systems and relates to each of these sub-fields of Artificial Intelligence to a certain extent [90]. In Chapter 1, we briefly discussed how many MAP configurations are possible depending on the nature of the problems. Different frameworks exist for different aspects of the problems, like when the agents are cooperative, strategic, or self-interested, whether the environment is stochastic or static, and when agents have different capabilities and their abilities to achieve something in the world is stochastic or non-stochastic. Table 2.1 shows a taxonomy of planning models based on different available settings. For more details, we refer readers to [87, 88, 90, 92–94].

Variants of the class of MAP problems targeted in this work are often represented using models like Dec-POMDPs [60], Qualitative Dec-POMDPs [18], MA-STRIPS [14], MA-PDDL [44, 45], and more. MAP researchers generally focus on devising domain-independent automated multi-agent planning algorithms to solve these models efficiently. In this work, we formalize a representation for planning for multiple agents with interacting actions, and, to support efficient planning using this formalism, develop domain-independent, efficient methods in centralized and distributed settings.

This chapter provides the background required for the thesis in detail, and we survey related work relevant for MAP in the desired settings. Some of the primary motivations behind developing this work and the general study of multi-agent systems [40, 93, 100] are to provide speed-up (via problem decomposition and distributed computation), scalability, and flexibility (via adding additional agents, changing their capabilities, or making problems harder), representation, and privacy. Hence we describe appropriate work from the literature regarding both automated planning and multi-agent system areas up to an extent. A more in-depth look into the work closely related to the new algorithms, formalisms, and results presented in this thesis is given in Chapters 3, 4 and 5.

Observability	**Actions**	**Single-Agent**	**Multi-Agent**	
			Non-Strategic	*Strategic*
Fully Observable	Det.	STRIPS (Classical Planning)	Factored Planning, MA-STRIPS, MA-PPP	Perfect Information Games
	NonDet.	MDP, Conformant Planning	Factored and Decentralized MDPs	Stochastic Games
Partially Observable	Det.	Contingent Planning	(Deterministic Qualitative) Decentralized POMDP	-
	NonDet.	POMDP	(Qualitative) Decentralized POMDP	POSG

Table 2.1 (Following [90]) Different planning models depending on particular settings: Abbreviations used in this table are: Det. - Deterministic; NonDet. - Non-Deterministic; MDP - Markov Decision Process; POMDP - Partially Observable MDP; STRIPS - Stanford Research Institute Problem Solver; MA-STRIPS - Multi-Agent STRIPS; POSG - Partially Observable Stochastic Games; MA-PPP - MA Privacy-Preserving Planning

2.1 Background

We describe each building block used to build this work (following Table 2.1), independently. We also present the problem models that this work uses or extends. Since automated planning is our main objective, we informally define it as the problem of sequencing a number of actions performed in an environment so that it is transformed from its current state to a state that satisfies some prestated goal conditions.

2.1.1 Single-Agent Frameworks

Before we delve into the details of multi-agent planning and decision-making frameworks, we first discuss relevant approaches and frameworks for single-agent planning problems succinctly. However, we assume that readers are familiar with these topics, and hence the upcoming definitions and formalisms would be only a refresher to introduce notations. For more details, we refer readers to other sources [35, 77]. In our work, we use multiple generalizations of these single-agent frameworks for accounting for multiple agents.

STRIPS Planning

The Stanford Research Institute Problem Solver, or in short STRIPS, is an automated planner developed by Fikes and Nilsson (1971) at SRI International [32]. Later, STRIPS also became the name for a formal language representing the classical planning problem, which works as an input to an automated planner.

Definition 1 *(Planning Task [32]). The STRIPS task, or* planning *in short, is a 4-tuple* $\langle P, A, I, G \rangle$ *such that its components are:*

- *P is a set of* conditions *(i.e., propositional variables or atoms). An element* $s \subseteq P$ *is considered a state of the system. The initial state,* $I \subseteq P$*, encodes a state that has a set of conditions that are true initially, and* $G \subseteq P$ *encodes the goal conditions that must hold for a state to be considered a goal state.*
- *A is the set of* actions *such that each action* $\mathrm{a} \in A$ *has syntax and semantics based on the standard STRIPS formalism, that is,* $\mathrm{a} = \langle \mathit{pre}(\mathrm{a}), \mathit{eff}^{+}(\mathrm{a}), \mathit{eff}^{-}(\mathrm{a}) \rangle$. *Here,* $\mathit{pre}(\mathrm{a})$ *is the precondition of the action such that for the action* a *to be applicable in a state* $s \subseteq P$*, its precondition should be* true *in the state, i.e.,* $\mathit{pre}(\mathrm{a}) \subseteq s$*, and,* $\mathit{eff}^{+}(\mathrm{a})$ *and* $\mathit{eff}^{-}(\mathrm{a})$ *are the positive effect and negative effect of this action, respectively:*

$$a(s) = (s - \mathit{eff}^{-}(a)) \cup \mathit{eff}^{+}(a) \tag{2.1}$$

If an action a *is applicable in* s*, we denote the resulting state as* a(s)*, while the resulting state is undefined if the action is not applicable in s (i.e.,* $\mathit{pre}(\mathrm{a}) \nsubseteq \mathrm{s}$*).*

Based on Definition 1, a *solution* to a STRIPS task is a plan (comprising a sequence of actions), $\pi = (a_1, a_2, ..., a_n)$, such that $G \subseteq a_n(...a_2(a_1(I))...)$. That means, the plan π transforms the initial state to a state satisfying the goal conditions.

Contingent Planning

We now define the standard contingent planning problem and its solution structure below, followed by some contingent offline and online solvers extensively used in this thesis.

Definition 2 *(Contingent Planning Task [1]). The Contingent Planning Task is also known as a planning problem with partial observability and sensing actions (PPOS). It is described as a quadruple* $\langle P, A, \varphi_I, G \rangle$*, where P is a set of propositions, A is a set of actions,* φ_I *is a propositional formula that describes the set of possible initial states, and* $G \subseteq P$ *denotes the goal propositions.*

For an atom $p \in P$, its corresponding positive and negative literals are p and $\neg p$, respectively. A set of literals is considered a *conjunction* of the literals as well as the *assignments* to the atoms in the set. For example, the set of literals $\{p, q, \neg r\}$ is treated both as the formula $p \wedge q \wedge \neg r$, and as an assignment of *true* to p, *true* to q, and *false* to r.

- A state s is the *truth* assignment to propositions $p \in P$. A *belief-state* is a set of states such that each state could potentially be the *true* state. The *initial* belief-state is, $b_I = \{s : s \models \varphi_I\}$, or in other words, b_I represents a set of all states that satisfy the formula φ_I, initially.

- An action $a \in A$ is represented as a 3-tuple, $\langle pre(a), \mathit{eff}(a), obs(a)\rangle$ such that,

  1. The set $pre(a)$ encodes a set of literals, and represents the precondition of the action.
  2. The set $\mathit{eff}(a)$ denotes a set of pairs of the form (c_i, e_i), and represents a set of conditional effects. Here, c_i is a conjunction of multiple literals, while e_i is a *single* literal. Suppose that the action a is executed in the state s; then $e_i \in a(s)$ if $c_i \subseteq s$. For more details, see Anderson, Smith, and Weld (1998) [3]. We assume that each action is *well-defined*, which implies the following: (a) When both (c_i, e_i) and (c_j, e_j) belong to the effect of an action and $s \models c_i \wedge c_j$, then the domain modeler would make sure that $e_i \wedge e_j$ is consistent. (b) If (c_i, e_i) belongs to the effect of this action, then $pre(a) \wedge c_i$ is consistent. For simplicity, we assume that all the actions are *deterministic* in nature, although these approaches can be adopted to handle non-deterministic actions, too. A *non-deterministic* action models multiple effects that are possible after its execution, and one particular effect cannot be determined prior to the execution.
  3. The propositions in the set *obs(a)* denote those propositions whose values will be observed once this action is executed. We assume that all the observations are deterministic, accurate, and immediate. They *reflect* the state of the world *before* the action a is executed.

In the benchmark problems, unless we specify explicitly, each action $a \in A$, is either a sensing action, (*i.e.*, $\mathit{eff}(a) = \emptyset$, hence truly a sensing action that does not change the state of the world), or a non-sensing action, (*i.e.*, $obs(a) = \emptyset$, hence an action that only changes the state of the world). However, this is just an assumption to make things simple and not a limitation of the formalism.

If an agent φ_i executes an action a and a proposition $p \in$ *obs(a)*, then following its execution, φ_i will *observe* p if in the current state p holds; otherwise φ_i will *observe* $\neg p$.

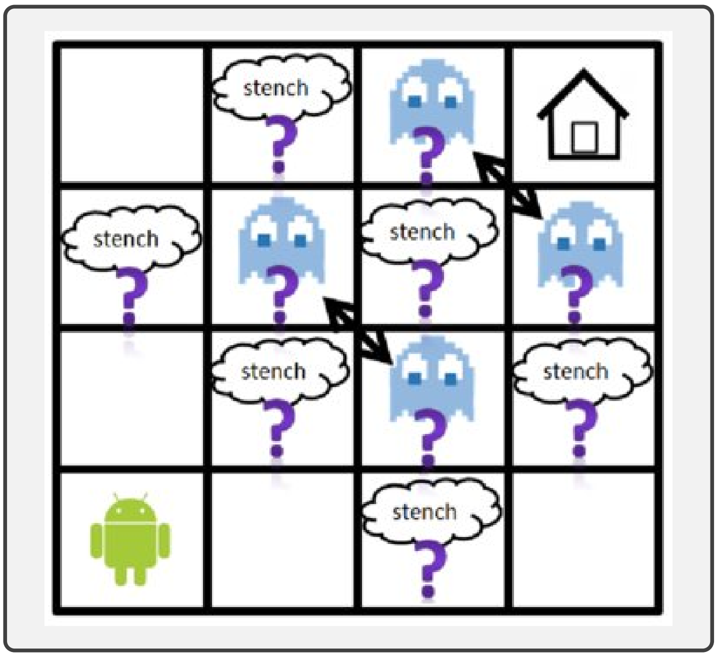

Fig. 2.1 The 4 × 4 Wumpus domain.

Therefore, for a given true state of the world s and the current belief state b, the resulting belief state once a is executed corresponds to progressions through this action a of the states in b that assign atoms in $obs(a)$ the same values as the state s does. That means, $b_{a,s} = \{a(s')|s' \in b,$ and s and s' agree on $obs(a)\}$.

Contingent Solution: The *solution* of a contingent planning task is represented by a *tree* (or *graph*) $\tau = (N,E)$. Each node N of this solution tree is labeled with an action $a \in A$ (or in other words, for each node N, $N.action$ returns $a \in A$), while the edges E represent the observations. The actions with no observations (*i.e.*, $obs(a) = \emptyset$), each possessing a *single-child*, are used to label nodes. An edge leading from any such node represents a *null* observation. Otherwise, a node in the plan tree has an edge corresponding to each observation value. This edge is labeled by the value of the corresponding variable observed.

The contingent plan is executed as follows: Starting at the root node, we execute the action associated with the node. As per our formalism, if this action is a non-observation action, we move to its (only) child (i.e., $N.single\text{-}child$, where N is the root node). Otherwise, we move along the edge labeled by the observed value to the appropriate child. Consider a simple scenario in which the root node is labeled by action a such that $obs(a) = \{p\}$ and p could either be *true* or *false*. The edge labeled with $p == false$ would lead to a *node* that is the *left-child* of the current node ($N.left\text{-}child$), while the other edge labeled by $p == true$ would

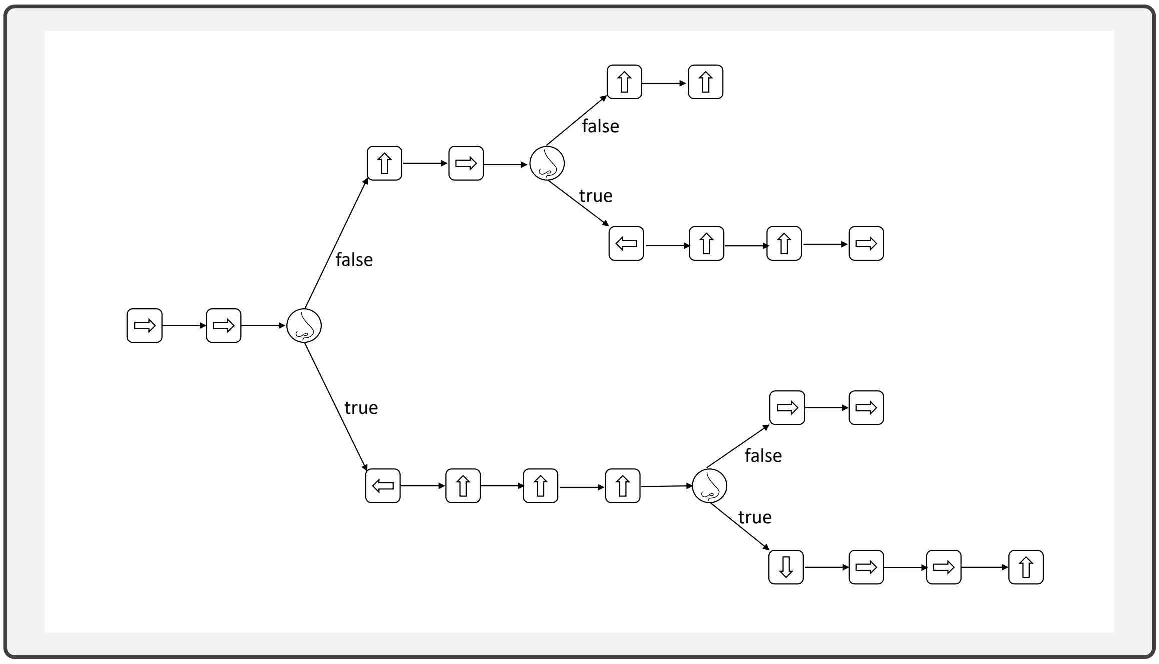

Fig. 2.2 A possible solution tree for the 4 × 4 Wumpus domain. Each arrow shows the direction in which the agent takes a move. Each of the decision nodes represents a *sensing* action (i.e, *smell(stench)*).

lead to a node that is the *right-child* of the current node (i.e., *N.right-child*). We continue recursively until a leaf has been reached. As we assume that actions and observations are deterministic, there is a single possible execution path along this tree for each initial state. We use $\tau(s)$ to denote the state obtained when τ is executed starting in state s. τ is a *contingent plan* (or *solution plan*) for the contingent planning task if $\tau(s) \vDash G$, for every $s \in b_I$.

The initial belief state (b_I) is associated with the root node of a solution tree, and is represented as n_{b_I}. Suppose that $n.b$ represents the belief state associated with the node $n \in N$ in the plan tree, and n is labeled by the action a. If n' represents the child node of n such that the directed edge from n to n' is labeled by an observation p, then the belief state n'_b associated with n' is, $\{a(s) : s \in b, s \vDash p\}$.

In Figure 2.1, we illustrate a contingent planning problem using a 4 × 4 Wumpus domain [1]. For the purpose of describing the terminologies, we discuss this example and show its one possible solution tree or contingent plan tree. The contingent planning problem is described as follows: An agent needs to navigate from the bottom-left corner (i.e., location [1,1]) to the top-right corner (i.e., the cell [4,4]). It is allowed to move in any of the four primary directions. There could be a monster called Wumpus either in the cell [2,3] or [3,2]. There could be another Wumpus either in the cell [4,3] or [3,4]. Wumpus has a specific property that it *emits* a stench that drifts to all adjacent cells. Hence, an agent can *smell* the stench if it is currently at a square adjacent to the current location of a Wumpus. However, the agent cannot determine in which adjacent cell the Wumpus is hiding. Therefore, smelling at more than one location might be required to know the precise position of the Wumpus. An agent can only move to a square if it is *safe*. Therefore, there are no dead-ends in the domain. One possible solution for this contingent problem is shown in Figure 2.2.

In general, a contingent plan may have a different branch, depending on every possible start state. Therefore, the total number of branches in a contingent plan tree can be exponential in the number of hidden state variables specified initially in the contingent problem. Reasoning directly about all these potential paths is very difficult in practice. However, if we know the true initial state, we can often find a solution quickly using a classical planner. And in practice, this solution is often effective for many other initial states. The CPOR planner exploits this idea, assumes a single state in the belief state is the “true” current state, and focuses on building one branch at a time in order to generate a complete contingent plan tree offline (described in [43]). By using an efficient classical solver to generate each path, and because many such path work for multiple initial states, CPOR is able to solve contingent problems of significant size.

The CPOR planner, which uses the SDR planner [81], is used as an underlying single-agent contingent planner in Chapters 3 and 4 for solving MAP problems with partial observability. Therefore, in this sub-section, the discussion is based on the following order: We will first describe the SDR planner, an online contingent replanner. SDR is used as a subroutine by the CPOR planner that calls SDR iteratively to build a full contingent plan tree. Then, we describe CPOR.

The SDR Planner

Sample, *Determinize*, and *Replan* (SDR) is an online contingent planner that uses replanning [16, 81]. The authors adapted popular approaches used in online MDPs to classical, non-stochastic domains that model sensing actions with partial observability. At a high level, SDR works as follows: Based on the original contingent problem, it *induces* a classical planning problem that is solved using a classical planner. Each action in this sequential plan is executed one by one, by the “agent” while it is *safe*, i.e., all its preconditions hold in every possible world. If an action is not safe to execute the algorithm replans.

Similar strategies have been used in other online planning models [105] and were extended by state sampling techniques, too [42, 106]. In SDR, the authors adapted this idea to domains that model partial observability and uncertainty about the initial state.

In principle, one could select a state from the given initial belief state and consider it the “true” initial state and plan using an off-the-shelf classical planner. However, while solving this induced classical problem, we note that the classical planner will have no incentive to insert sensing actions. There is no reason to take a more costly path that applies a pure sensing action that has no effect on the state of the world. Therefore, SDR uses a knowledge-based translation approach that translates contingent planning to classical planning [65]. The translation models and reasons about the agent's *knowledge* of the world, implicitly, rather than the real state of the world. The goal now becomes that the agent must *know* that the goal has been accomplished. SDR also uses a state sampling technique to overcome the shortcomings associated with Palacios and Geffner's translation (that has scalability issues as the classical planning problems become too complex to solve for the state-of-the-art classical planner).

The SDR approach works as follows. An initial state s_i is sampled from b_I. The planner uses this initial state to *determinize* the effect of sensing actions (required for the problem to be translated into a classical planning problem). It then uses the knowledge-based translation to generate a plan and follows this plan online. It follows this plan until either an unexpected observation is made (i.e., one that would not be possible if s_i was the true initial state), or until the planner cannot prove (using regression) that the preconditions of the next action must hold. At this point, the planner replans from the current belief state.

Contingent Planning using Online Replanning (CPOR)

We already mentioned that the solution to the contingent planning problem is represented as a plan tree, which branches on different possible observations as shown in Figure 2.2. Recent offline contingent planners like PO-PRP [55] use a translation-based approach that translates the original contingent planning problem into a fully observable, non-deterministic planning problem. Then they use non-deterministic planning [10]. Although this approach is successful, the translation may become very large, making the translated problem very difficult to be solved for a non-deterministic planner. In the past, the planners generating complete contingent plan trees encountered scaling-up issues, too [1, 20].

Orthogonal to these algorithms is the state-of-the-art offline contingent planner CPOR. CPOR uses SDR repeatedly, offline, to generate a full contingent plan tree for online execution [43]. The advantage of using an online planner is that it plans just for the next action (or, more accurately, until the next *sensing* action). When it makes an observation, it replans based on the information obtained. This approach can be used to generate a single branch of a complete plan tree, from its root node to one leaf node, corresponding to a particular set of initial states that yield identical observations. That is, the offline planner selects an initial state s, and simulates the online planning process, when s is the true initial state. CPOR starts simulating the planning process in the environment such that for each observation possible using the next sensing action in the plan, the process calls the SDR solver. Note that this whole process repeats itself until all branches end in goal leaves.

The CPOR algorithm works as follows (at a high-level): It starts with an empty stack S. The root node (n_0) of the plan tree corresponds to the initial belief state b, such that, initially, S contains the root node n_0. In each iteration, a node n is popped from S. An online replanner is called for n, considering the belief state associated with n as the current belief state. SDR returns a sequence of actions that either reach the goal or until the next sensing action. For each action, a in the sequence the following is done: first, its applicability in the current belief state is verified, and later, the next node, n', of the plan tree is generated - both in a principled way. If the sequence ends up with a sensing action, then the children of this sensing action will be pushed back to S. This process repeats until $S = \emptyset$ and returns the plan tree.

We note that CPOR does not backtrack. Also, since it uses an online solver to generate an offline solution, it is prone to getting into a dead-end during execution. Therefore, CPOR is incomplete; however, for a given sound and complete online replanner, and if we restrict ourselves to only deterministic domains with no dead-ends, the CPOR planner produces sound and complete contingent plans. For more details and to understand several optimizations used in the paper to improve its efficiency, we refer readers to (Komarnitsky and Shani 2016).

2.1.2 Multi-Agent Frameworks

We review relevant multi-agent frameworks, some of which are direct extensions (or a generalization to the MA setting) of the single-agent frameworks reviewed in the previous section upon which this work is developed.

Multi-Agent STRIPS

The MA-STRIPS framework is a *minimalistic* extension of STRIPS, in which the actions modeled in the STRIPS domain get distributed into subsets that correspond to multiple agents [14].

Definition 3 *A Multi-Agent STRIPS planning task that is also known as a multi-agent planning task or MA-STRIPS, for a planning system consisting of multiple decision makers* $\Phi = \{\varphi_i\}_{i=1}^{n}$, *is represented by a 4-tuple* $\Pi = \langle P, \{A_i\}_{i=1}^{n}, I, G\rangle$, *where*

- *P, I, and G are, respectively, the set of propositions, the initial state, and the goal conditions (they are the same as in* STRIPS*).*

- *The variable i represents the index of agent φ_i. There are n agents, while each subset A_i contains the actions that the agent φ_i can perform. Each action $a \in A_i$ follows the standard STRIPS syntax and semantics.*

Its Solution: A solution for the MA-STRIPS task is a sound plan if and only if when the identities of the agents in this plan are *masked*, this plan is a solution plan for the original underlying STRIPS problem, too. It is trivial to see that for $n = 1$, MA-STRIPS will reduce to STRIPS. However, the assumption in this simplistic extension is that action sets A_i, for $i = 1$ to n, are disjoint, and $\bigcup_{i=1}^{i=n} A_i$ is equal to the complete set of actions.

The original work by Brafman and Domshlak (2008) that introduced MA-STRIPS [14] ignited a lot of interest in MAP, in general. They showed that the complexity of MAP based on the MA-STRIPS formalism is not exponential in the number of agents present, but proportional to the tree-width of their action interaction graph and the minimal number of interactions required to solve the MAP problem. They showed that this approach works quite well for loosely-coupled planning domains, e.g., the Logistics domain.

We briefly describe some relevant planners that used the MA-STRIPS formalism or its generalizations. Their detailed description is out of the scope of this work. Planning First [59] and MAD-A* [57] were the earliest planners based on this formalism. They dealt with state-space planning. Another approach was proposed based on plan-space planning extending the MA-STRIPS formalism, called MAP-POP [78], which later evolved into MAPF [97]. These planners used a generalization of MA-STRIPS.

In the following years several state-space based coordination planners were introduced, like MAPlan (appeared in CoDMAP 2015 planning workshop at ICAPS), MADLA Planner [91], and the Macro-MAFS planner [51]. Several plan-space planning based coordination planners, like PSM-VRD (based on Planning State Machines (PSMs)) [98], and hybrid coordination planners that combine elements of state-space and plan-state planners, like GPPP [53] and PP-LAMA [52], were also introduced. Some of these planners also deal with privacy-preserving planning in which, during planning, agents do not reveal their private information and often plan in a distributed manner.

Privacy in Multi-Agent Planning

Privacy guarantees in multi-agent planning are provided in the form of *private variables*, *private values*, and *private actions* [13]. If only one agent is aware of a *variable* or the existence of some *value*, then these are known to be *private* to the agent, while, if some *action* is private to an agent, (ideally) only this agent is aware of the existence of this action, its form, and its cost, etc.

A variable (or proposition) is said to be *private* to an agent φ_i, if this variable appears in the preconditions or effects of the actions of φ_i *only*. We consider a variable *private* only if all its values are private to an agent φ_i, or else, it is *public*. An action is private to this agent *iff* both preconditions and effects of this action are *private*. Actions not falling in this category are *public*. Note that a public action of an agent may also contain private preconditions and effects in the schema.

Let us describe the privacy guarantees associated with an algorithm. We say that an algorithm is *weakly private* if no agent communicates a private value of a variable, anywhere throughout the initial state, the goal, or an intermediate state, to an agent for which this value is not private during a run of the algorithm, and if the only description of its actions it needs to communicate to another agent, φ_j is in their public view.

In Secure Multi-Party Computation (MPC) [104], which is a sub-field of Cryptography, *stronger privacy* guarantees are sought from multi-party algorithms. Assume that we are in an ideal world. Further, considering a secure network, a trusted third-party may receive inputs from different parties, such that it performs required calculations and returns the solutions to the respective parties. Secure MPC studies how we can achieve this even if things are not ideal. Or in other words, the goal of methods for Secure MPC is to enable multiple agents to compute a function over their inputs while keeping these inputs private.

An algorithm is *strongly private* if no agent can deduce information like the existence of a value or a variable that is private to another agent, φ_i, or its private actions' models, which is beyond the information that can be deduced from the description of φ_i's actions, the public view of other agents' actions, and the public view of the solution plan [58]. More specifically, we will say that a variable (or some specific value of a variable) is *strongly private* if the other agents cannot deduce its existence from the information available to them [13]. It can also be rephrased as: "*If by execution of an algorithm, the agents do not obtain and cannot deduce any private information additional to what can be deduced only from the public input and public output of the algorithm, the algorithm is strong privacy-preserving.*"

A formal approach to multi-agent privacy using MA-STRIPS is proposed by Nissim and Brafman (2014), and later extended by Brafman (2015), loosely based on the standard concepts of Secure MPC. First, the authors specify private information the agents are attempting to hide, and later, they present two degrees of privacy preservation - weak and strong privacy.

Greedy Privacy-Preserving Planner: Maliah, Shani, and Stern (2017) devised an algorithm for privacy-preserving planning called the Greedy Privacy-Preserving Planner (GPPP) [53]. It generates an abstract and approximate global plan collaboratively for coordinating agents' actions, and later, this global plan is extended by the agents individually to make it executable.

At a high level, GPPP works as follows. The algorithm considers a relaxed version of the original MA planning problem, solved by all the agents collaboratively, and whose solution contains the coordination scheme. This global plan is then extended by each agent, independently, to an executable plan such that each agent focuses on their private information only. First, the algorithm builds a global plan for proper coordination among the agents. Then, each agent extends this global plan to an executable plan, by adding the missing private actions.

On the one hand, GPPP follows "coordination and planning", which can be seen as an algorithm similar to algorithms like "CSP+Planning" [15] and "Planning First" [59]. While on the other hand, GPPP also first plans to generate the global plan and coordinate things when needed, which can relate to "planning and coordination" like in Nissim and Brafman (2014) [58].

To enhance its overall performance, GPPP uses two privacy-preserving heuristics based on two well-known classical planning heuristics: Pattern Databases (PDBs [28]) and Landmarks (LMs [73]). The new heuristics are called privacy-preserving PDBs and privacy-preserving Landmarks, respectively, which are agnostic to this algorithm and can be used by other privacy-preserving algorithms. They show the benefits of using these heuristics and the advantage of GPPP over contemporary privacy-preserving planners for the multi-agent STRIPS formalism.

The ADP Planner

Crosby, Jonsson, and Rovatsos (2014) (hereafter, CJR) proposed a different method to solve the Multi-Agent STRIPS problem [23]. Their approach handles the MA planning problem based on explicitly specified constraints on an object set in a multi-agent domain. This constraint on the object set allows or restricts concurrent execution of the actions of the entities such that these actions can manipulate this set in a certain way. In other words, CJR's object cardinality constraints constrain the set of *legal* joint actions. An example of such a constraint could be the independent move actions of two robots, where the robots can move concurrently but *cannot* occupy the same location, simultaneously. Another example is that a *minimum* of two agents is needed to sail a boat in the river.

For planning, CJR suggest a compilation-based approach that compiles MA problems, which explicitly specifies cardinality constraints to allow or to restrict specific types of actions and their numbers to *manipulate* an object set, to classical planning problems that can be solved using an off-the-shelf classical planner. Their method post-processes the resulting plan and compresses it by allowing actions that do not conflict with each other to be executed concurrently. The resulting planner is called the ADP planner, which is also a non privacy-preserving planner. In another compilation-based approach, similar to CJR's, Crosby and Petrick (2014) propose to encode *affordances* (they represent object-action tuples) and determine the conditions under which an object can (or must) be manipulated concurrently. This approach solves a planning domain with such encoding by translating it to a temporal planning domain [24]. In our work, we describe a planning formalism that handles planning with interacting actions in Chapter 5; this is a generalization of CJR's approach. It solves some problems that cannot be solved by their formalism.

Qualitative Decentralized POMDPs

Contingent planning deals with a single agent in an environment under uncertainty with partial observability. However, as it considers only a single-agent, it does not model the effect of uncertainties caused by the presence of multiple acting entities in this environment. This is handled by its multi-agent extension, known as a *Qualitative Decentralized* POMDP (QDec-POMDP). The QDec-POMDP framework is a conceptually simpler version of the famous Dec-POMDP framework (which we discuss in greater detail in the Related Work Section (Section 2.2)). Dec-POMDPs provide a rich, attractive model for MA planning under uncertainty with partial observability in cooperative settings with a growing body of research [7, 79], where the NEXP hard complexity of solving Dec-POMDPs has limited its scalability to larger problems and its applicability, too.

The QDec-POMDP framework is a qualitative propositional model for MAP under uncertainty with partial observability. It has a non-probabilistic structure and is introduced as an alternative model to Dec-POMDPs. Although the two frameworks share a similar worst-case complexity, the QDec-POMDPs model has several advantages, e.g., being geared to a propositional state model, its specifications is easier compared to a flat representation typically used for Dec-POMDPs. QDec-POMDP planning is also more *classical* in nature, and hence easier to describe, and it also eases the adaptation of recent advancements of classical planning frameworks and heuristics. This is especially true when its deterministic variant is used. Therefore, it allows us to solve much larger problems than the current Dec-POMDP algorithms can handle.

Model Definition: We start with the basic definition of a flat-space *Qualitative Decentralized* POMDP, followed by a factored formalism motivated by contingent planning model definitions [11, 17], which we will be using in our work.

Definition 4 *A qualitative decentralized partially observable Markov decision process (QDec-POMDP) is a tuple $Q = \langle I, S, b_0, \{A_i | i \in I\}, \delta, \{\Omega_i\}, O, G\rangle$, where*

- *I is a finite set of agents indexed $1,...,m$. We often refer to the i^{th} agent as φ_i (i represents its index).*
- *S is a finite set of states.*
- *$b_0 \subset S$ is the set of states initially possible.*
- *A_i is a finite set of actions available to agent φ_i, and $\vec{A} = \otimes_{i \in I} A_i$ is the set of joint actions, where $\vec{a} = \langle a_1, ..., a_m\rangle$ denotes a particular joint action. (It is assumed that $\vec{A} = \otimes_{i \in I} A_i$, where each A_i is the set of actions of agent φ_i. One can also identify the action $a_i \in A_i$ with the joint action $\langle noop_1, \ldots noop_{i-1}, a_i, noop_{i+1}, \ldots, noop_m\rangle$.*
- *$\delta : S \times \vec{A} \rightarrow 2^S$ is a non-deterministic Markovian transition function. $\delta(s, \vec{a})$ denotes the set of states that can be reached when taking joint action $\vec{a}$ in state s.*
- *Ω_i is a finite set of observations available to agent φ_i and $\vec{\Omega} = \otimes_{i \in I} \Omega_i$ is the set of joint observation, where $\vec{o} = o_1, ..., o_m$ denotes a particular joint observation.*
- *$\omega : \vec{A} \times S \rightarrow 2^{\vec{\Omega}}$ is a* non-deterministic *observation function. $\omega(\vec{a}, s)$ denotes the set of possible joint observations $\vec{o}$ given that joint action $\vec{a}$ was taken and led to outcome state s. Here $s \in S$, $\vec{a} \in \vec{A}$, $\vec{o} \in \vec{\Omega}$.*
- *$G \subset S$ is a set of goal states.*

For this formalism, we do not assume a finite *horizon* for limiting the maximal number of actions in each execution. We focus, however, on deterministic outcomes and deterministic observations. In such cases, a successful solution is acyclic, and hence there is no need to bound the number of steps. Extension to domains with non-deterministic outcomes with a bounded horizon is left for future work. In this work, we assume a shared initial belief, like most Dec-POMDP models, which is most natural for an off-line centralized algorithm (again, like most Dec-POMDP algorithms).

We will work with a factored representation of QDec-POMDP, which is specified using the following components: $\langle I, P, \{A_i | i \in I\}, Pre, Eff, Obs, b_0, G'\rangle$ where I is a set of agents, P is a set of primitive propositions, $\vec{A}$ is a vector of individual action sets, Pre is the precondition function, Obs is an observation function, Eff is the effects function, b_0 is the initial state formula, and G is a set (conjunction) of goal propositions.

We now explain the QDec-POMDP induced by such factored QDec-POMDP specifications. First, its state space, S, consists of all truth assignments to P, and each state can be viewed as a set of literals. Its initial states and goals are all states that satisfy the initial state formula and the goal conjunction, respectively.

The transition function δ of the QDec-POMDP model is defined using the *Pre* and *Eff* functions, described in the following paragraphs.

The precondition function *Pre* maps each individual action $a_i \in A_i$ to its set of preconditions, i.e., a set of literals that must hold whenever agent φ executes a_i. Preconditions are local, i.e., defined over a_i rather than $\vec{a}$, because each agent must ensure that the relevant preconditions hold prior to executing its part of the joint action. We extend *Pre* to be defined over joint actions $\{\vec{a} = \langle a_1, .., a_m \rangle : a_i \in A_i\}$ (where $m = |I|$): $Pre(\langle a_1, .., a_m \rangle) = \cup_i Pre(a_i)$.

Brafman, Shani, and Zilberstein (2013) define an effects function *Eff* mapping joint actions onto a set of pairs (c, e) of conditional effects, where c is a conjunction of literals and e is a single literal, such that if c holds before the execution of the action, e holds after its execution. The effects are a function of the joint action rather than the agents' local actions, as can be expected, due to possible interactions between the agents' local actions.

We assume that single-agent actions that are executed concurrently do not interact, unless specified explicitly. Such interactions are then modeled by collaborative actions. Collaborative actions have the same form as single-agent actions, except that they have multiple agent parameters. Thus, an agent may have a single-agent *move* action, as well as participating in a collaborative, two-agent action, *joint-lift*, for lifting a table. One can think of *joint-lift* as two concurrent single-agent *lift* actions (e.g., as modeled in [82]). If a collaborative action such as *joint-lift* exists, and a single-agent *lift* exists, too, then it is forbidden for the planner to schedule two separate single-agent *lift* actions at the same time. If it wishes to perform the two *lift* actions concurrently, it must use the *joint-lift* action. For a deeper discussion of the issue of defining joint actions, see [82, 83]. We remark here that when one does not allow concurrent actions that delete one another's preconditions or object capacity constraints [23], then one can consider only joint actions that consist of a single (possibly collaborative) action at each step with all other agents performing no-ops, which greatly simplifies the process. Later, the plan can be made more compact in post-processing, e.g., using the technique of Crosby, Jonsson, and Rovatsos (2014).

For every joint action $\vec{a}$ and agent φ_i, the observation function is $Obs(\vec{a}, \varphi_i) = \{p_1, \dots, p_k\}$, where $p_1, ..., p_k$ are the propositions whose value agent φ_i observes after the joint execution of $\vec{a}$. The observation is private, i.e., each agent may observe different aspects of the world. We assume that the observed value is correct and corresponds to the post-action variable value. In our domains, for each individual action set A_i, we will separate actions into observation and non-observation actions. The former do not affect the world state, and the latter have an empty set of observations. Note that this is just an assumption to make things simple and not a limitation of the formalism. Every action can be separated into a non-observation and an observation action by adding suitable propositions forcing the two to appear consecutively in every plan.

While QDec-POMDPs allow for non-deterministic action effects as well as non-deterministic observations, we focus in this work only on deterministic effects and observations, and leave discussion of an extension of our methods to non-deterministic realms to future research.

Joint Policy (*QDec's solution*): We can represent the local plan of an agent φ_i using a *policy tree* τ_i, which is a tree with branching factor $\leq |\Omega|$. The nodes of this tree are labeled by actions. The edges that follow a sensing action are labeled by *observations* – one edge for each. To execute the plan, each agent performs the action at the root of the tree and then uses the subtree labeled with the observation it obtained for future action selection. If τ_i is a policy tree for agent φ_i and o_i is a possible observation for agent φ_i, then $\tau_{i_{o_i}}$ denotes the subtree that is rooted by the child of the root of τ_i that is reached via a branch labeled by o_i (Figure 2.2 can be considered as a reference to τ_i).

Let $\vec{\tau} = \langle \tau_1, \tau_2, \cdots, \tau_m \rangle$ be a vector of policy trees, also called a *joint policy*. We denote the joint action at the root of $\vec{\tau}$ by $\vec{a}_{\vec{\tau}}$, and for an observation vector $\vec{o} = o_1, \ldots, o_m$, containing each agent's observation, we define $\vec{\tau}_{\vec{o}} = \langle \tau_{1_{o_1}}, \ldots, \tau_{m_{o_m}} \rangle$.

Since in QDec-POMDPs (unlike in Dec-POMDPs), an action may have preconditions, a joint policy tree is executable only if the preconditions of each action hold prior to its execution. To check this, we must maintain the sets of states possible at each point in time during the execution of the joint policy. This is usually referred to as the *belief state*. Notice that this is the belief state of the entire system, not of a single agent. Online, each agent will have less information, because it cannot distinguish between all branches of the joint policy. However, here we are taking the point of view of the off-line planner. To follow policy $\vec{\tau}$, we first consider the action $\vec{a}_{\vec{\tau}}$ given the current belief state b. It must be the case that $b \models pre(a_\tau)$. In that case, we say that $\vec{a}_{\vec{\tau}}$ is *executable* in b. After the agents execute a_τ and observe $\vec{o}$, their new belief state is $tr(b, \vec{o}, \vec{a}_\tau) = \{a_\tau(s) | s \in b, a_\tau(s) \models \vec{o}\}$.

We say that a joint policy $\vec{\tau}$ is *executable* given the initial belief state b if (1) $\vec{a}_{\vec{\tau}}$ is executable in $\vec{b}$; (2) if a_{τ_i} is a part of a collaborative action and φ_j is another agent participating in that collaborative action, and then a_{τ_j} contains φ_j's part of that collaborative action; and (3) For every possible joint observation $\vec{o}$, $\vec{\tau}_{\vec{o}}$ is executable given $tr(b, \vec{o}, \vec{a}_{\vec{\tau}})$.

A joint policy is called a *solution* if it is executable, and for all leaf nodes in the tree $\bigcap_i b_i \models G$, i.e., the set of possible states given the joint local beliefs of the agents satisfy

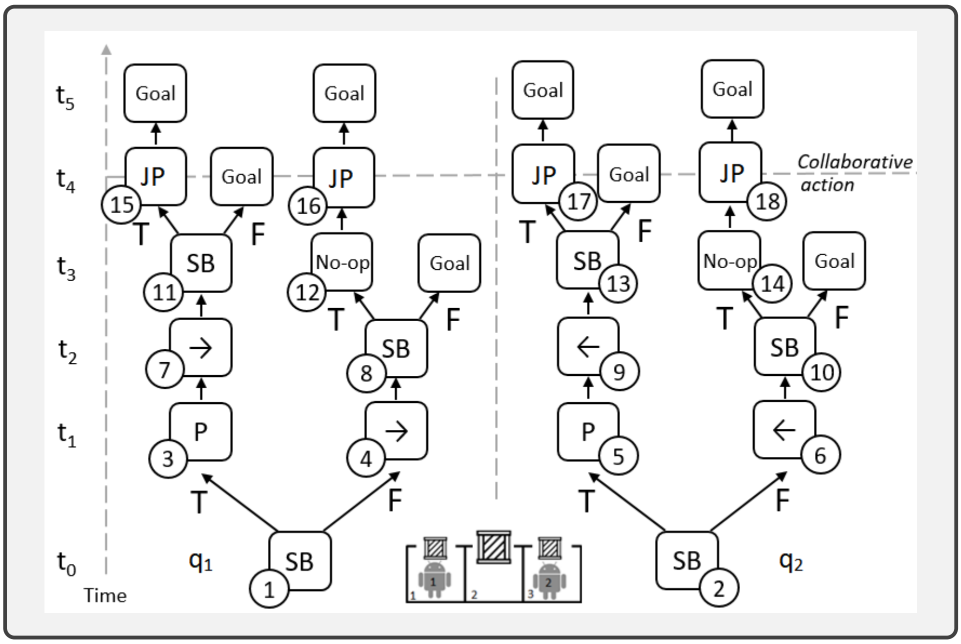

Fig. 2.3 Illustration of Example 1 showing the Box-Pushing domain with 2 agents and a possible set of local plan trees that produce a solution. Possible agent actions are sensing a box at the current agent location (denoted *SB*), moving (denoted by arrows), pushing a light box up alone (denoted *P*), jointly pushing a heavy box (denoted *JP*), and no-op.

the goal. Note that unlike Dec-POMDPs, for QDec-POMDPs there is no obvious notion of optimal policy, or an optimization criterion, although one could strive to find trees with smaller depth, or trees that minimize the maximal branch cost.

In Chapter 1, we used an example in Figure 1.1 to shows a solution, a joint-plan tree for a problem in the Box-Pushing domain. We now illustrate the factored QDec-POMDP model using the same problem, in Example 1, while its solution comprising individual plan trees for each agent is depicted in Figure 2.3.

Example 1 *In this example there is a 1D grid of size 3, with cells marked 1-3, and two agents, starting in cells 1 and 3. In each cell there may be a box, which needs to be pushed upwards. The left and right boxes are light, and a single agent may push them alone. The middle box is heavy, and requires that the two agents push it together.*

We can hence define $I = \{1,2\}$ *and* $P = \{AgentAt_{i,pos}, BoxAt_{j,pos}, Heavy_j\}$ *where* $pos \in \{1,2,3\}$ *is a possible position in the grid,* $i \in \{1,2\}$ *is the agent index, and* $j \in \{1,2,3\}$ *is a box index. In the initial belief state each box may or may not be in its corresponding cell —* $b_0 = AgentAt_{1,1} \wedge AgentAt_{2,3} \wedge (BoxAt_{j,j} \vee \neg BoxAt_{j,j})$*, for* $j = 1,2,3$*. There are therefore 8 possible initial states.*

The allowed actions for the agents are to move left and right, to push a light box up, or jointly push a heavy box up with the assistance of the other agent. There are no preconditions for moving left and right, i.e., $Pre(Left) = Pre(Right) = \phi$. *For agent* φ *to push up a light box* j, *agent* φ *must be in the same place as the box. That is,* $Pre(PushUp_{i,j}) = \{AgentAt'_{i,j} \neg Heavy_j, BoxAt_j\}$. *For the collaborative joint push action the precondition is* $Pre(JointPush_j) = \{AgentAt_{1,j}, AgentAt_{2,j}, Heavy_j, BoxAt_j\}$.

The moving actions transition the agent from one position to the other, and are independent of the effects of other agent actions, e.g., $Right_i = \{(AgentAt_{i,1}, \neg AgentAt_{i,1} \wedge AgentAt_{i,2}), (AgentAt_{i,2}, \neg AgentAt_{i,2} \wedge AgentAt_{i,3})\}$. *The only joint effect is for the JointPush action —* $Eff(PushUp_{1,2}, a_2)$ *where* a_2 *is some other action, are identical to the independent effects of action* a_2, *while* $Eff(PushUp_{1,2}, PushUp_{2,2}) = \{(\phi, \neg BoxAt_{2,2})\}$, *that is, if and only if the two agents push the heavy box jointly, it (unconditionally) gets moved out of the grid.*

We define sensing actions for boxes — $SenseBox_{i,j}$, *with precondition* $Pre(SenseBox_{i,j}) = AgentAt_{i,j}$, *no effects, and* $Obs(SenseBox_{i,j}) = BoxAt_{j,j}$. *The goal is to move all boxes out of the grid, i.e.,* $\bigwedge_j \neg BoxAt_{j,j}$.

2.2 Related Work

This section presents a review of related work, highlighting where our research resides on a big chart depicting complete multi-agent planning research. For reference see Table 2.1. We give an overview of relevant frameworks applicable for MAP in an environment that is partially observable and, unlike in *Qualitative* Dec-POMDPs, stochastic, too. The Dec-POMDP framework offers a rich model for capturing MAP problems in the cooperative setting, and extends the single-agent POMDP model [22, 41] to accommodate multiple agents with possibly different information states. The POMDP framework is rich enough to model many realistic centralized problems with uncertainty and partial observability, except that it cannot handle multiple agents and uncertainty caused due to their presence in the environment.

2.2.1 Single-Agent Decision Frameworks

Before we dive into the core concepts of multi-agent planning and decision-making, we give concise descriptions of the single-agent models which the relevant multi-agent models extend.

Markov Decision Processes (MDPs)

A Markov decision process is a decision-theoretic, discrete-time stochastic control process that provides a probabilistic mathematical framework to model planning problems [6, 69, 70]. It models an agent interacting with an environment where the outcomes of an action performed are only partly controlled by the entity, and the rest is probabilistic. Formally, a Markov decision process is defined as follows.

Definition 5 *A Markov decision process is a 4-tuple $\langle S, A, Tr, R\rangle$, where*

- *S is a set of states, also known as state-space.*
- *A is a set of actions available to the entity.*
- *Tr represents the transition model of a stochastic system. Usually, it is provided as a table with entries of the form: $Tr_a(s,s') = Tr(s_{t+1} = s'|s_t = s, a_t = a)$. Here, t represents a* time-step *explicitly. The Markovian property lets only the current state influence the transition function, i.e., $Tr(s_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = Tr(s_{t+1}|s_t, a_t)$.*
- *R models the immediate reward (or maybe expected immediate reward). It is also represented as a table with entries of the form $R_a(s,s')$, which represents the reward received immediately for taking the action a in the state s and reaching the state s'.*

A solution of an MDP is a policy that is optimal according to some optimality criterion. We often differentiate between finite- and infinite-horizon policies. Optimal finite-horizon policies are non-stationary and are described e.g., using a sequence of decision rules, δ_t, one for each stage, such that $0 \leq t \leq (h-1)$, where h is the horizon. Each decision rule is a mapping from *state* to *action*. Infinite-horizon policies can be restricted to have a stationary form $\pi = (\delta)$ that is indifferent to the stage [77].

Partially Observable MDPs

A partially observable Markov decision process (or POMDP [22, 41]) is a mathematical framework that models partial observability, allowing the entity to have a set of observations and adding an observation probability table to the MDP (see Definition 5). The entity cannot sense its environment (or, for that matter, its current state) perfectly. Hence, the model must maintain a probability distribution over a set of possible true states based on that allowed set of observations and the probabilities from the provided table and the underlying MDP. The POMDP model is general enough to model many real-world scenarios like robot navigation problems, planning under uncertainty, etc. Formally, the POMDP model is defined as follows.

Definition 6 *A discrete-time Partially Observable MDP models the relationship, mathematically, between an entity and its environment. Formally, the POMDPs model is a 7-tuple* $\langle S, A, Tr, R, \Omega, O, \gamma \rangle$*, where*

- $S = \{s_1, s_2, ..., s_n\}$ *is a set of possible states.*
- $A = \{a_1, a_2, ..., a_m\}$ *is a set of actions available to the entity. An action can change the state of the world or (and) provide some unknown information about the world.*
- *Tr is the transition function (set of transition probability of the form* $Pr(s'|s, a)$*).*
- *R is the immediate reward function.*
- $\Omega = \{o_1, o_2, ..., o_k\}$ *is a set of observations available to the entity.*
- *O is an observation function describing the likelihood of sensing a specific observation* o *following an action* a *(i.e.,* $O(o|a, s')$ *– o is sensed after* reaching s'*).*
- γ *is the discount factor.*

In the basic setting, at each decision epoch, the entity takes an action a that changes the current state s (supposing that the environment is in state s currently) to the next state s' with a probability $Tr(s'|s, a)$. The entity immediately receives a reward $R(s, a)$ for executing a in s. Later, after arriving in s', it also receives an observation $o \in \Omega$ with a probability $O(o|a, s')$. This observation might change the current belief of the entity about the environment. This whole process repeats several times.

Like in MDPs, here, too, the objective is to find a policy that maximizes its expected total reward over the horizon.

A solution to a POMDP problem is a *policy* that assigns the next *action* to execute to each *actions-observations* history, or to each belief state. The former policy is usually represented as a *tree* or a *policy-graph* (i.e., also known as a finite state controller).

2.2.2 Multi-Agent Decision Frameworks

Even though the single-agent decision-making frameworks described above treat state uncertainty (and partial observability in POMDP) in a principled manner, they provide stochastic frameworks limited to single-agent settings. Hence, they do not capture uncertainties caused due to the presence of other decision makers in the world.

Decentralised POMDPs

The Dec-POMDP framework is a generalization of POMDPs, which is equivalent to Pynadath and Tambe's MTDP (*Multi-agent Team Decision Problem* [71]) framework. MTDP models a team of non-strategic agents, who intend to do some joint task, and studies possible policies of their behavior. Its underlying component has been taken from the initial decision model for the team [38], which is extended to handle decision making in a dynamic environment. Likewise, Dec-POMDPs allow for modeling a team of multiple cooperative agents situated in a stochastic environment that is partially observable, too. Let us first formally define the standard Dec-POMDP framework.

Definition 7 *A Dec-POMDP with* n *agents is represented as a tuple* $\langle I, S, A, T, R, \Omega, O, b_0, h \rangle$, *where*

- $I = \{1, 2, 3, ..., n\}$ *is the set of agents (such that each* $i \in I$ *represents the index of an agent).*

- S *is a finite set of states.*

- $A = \times_{i \in I} A_i$ *is the set of joint actions, while* A_i *is the set of actions available to the agent i. At each time step, agents together execute a joint action* $\vec{a} = \langle a_1, a_2, ..., a_n \rangle$, *but they do not sense each other's actions.*

- T *represents the transition function specifying transition probability, i.e,* $P(s'|s, \vec{a})$.

- $\Omega = \times_{i \in I} \Omega_i$ *is the set of joint observations, while* Ω_i *is the set of observations available to the agent i. At every stage the agents receive the joint observation* $\vec{o} = \langle o_1, o_2, ..., o_n \rangle$. *Here, each agent i observes only its own component* o_i *of* $\vec{o}$.

- O *is the observation function that specifies the joint-observation probability.* $P(\vec{o}|\vec{a}, s')$

- R *is the reward function. It maps states and joint actions to real numbers:* $R(s, \vec{a}) \in$ R.

- b_0 *represents the initial belief state, (generally) common to all the* n *agents. It is a probability distribution over a finite set of states, i.e.,* $b_0 \in P(S)$ *at* $t = 0$, *such that* $P(S)$ *is the infinite set of probability distributions over a finite set S.*

- h *is the horizon.*

At each stage $t = 0, 1, 2, ..., h - 1$, all agents together take a joint-action and receive a joint-observation.

The *task* is to find a joint-policy $\vec{\pi} = \langle \pi_1, \pi_2, ..., \pi_n \rangle$, which consists of individual policies, π_i for each agent i. At each step t, following its individual policy π_i, each agent chooses an action a_i as prescribed by $\pi_i : \times_{t-1}(A_i \times O_i) \rightarrow A_i$, or in other words, its own *actions* taken and *observations* received until time t. The joint goal of the agents is to maximize the future discounted reward, i.e., $V^{\vec{\pi}} = E_{\vec{\pi}}[\sum_{t=0}^{h} \gamma^t R]$, where γ is the discount factor.

Unfortunately, computing an optimal joint policy $\vec{\pi}^*$ for POMDPs is NEXP complete [7]. For a detailed review of Dec-POMDPs, see [7, 61].

It is common to solve Dec-POMDPs centrally, but the agents must execute their policies in a decentralized manner. That means that agents act independently based on the observations received in the environment, and assuming no explicit communication channels through which agents share information during execution. However, Dec-POMDPs can be modeled to support communication – which is often achieved implicitly through actions, observations, and states.

2.2.3 Decentralized Tiger Problem

To illustrate the Dec-POMDP model, we describe a surprisingly difficult toy example called the Dec-Tiger problem – a generalization of a well known toy domain for single-agent POMDPs [41]. This domain is often used as a Dec-POMDP benchmark [56].

$\vec{a}$	s_l	s_r
$\langle a_{Li}, a_{Li} \rangle$	-2	-2
$\langle a_{Li}, a_{OL} \rangle$	-101	+9
$\langle a_{Li}, a_{OR} \rangle$	+9	-101
$\langle a_{OL}, a_{Li} \rangle$	-101	+9
$\langle a_{OL}, a_{OL} \rangle$	-50	+20
$\langle a_{OL}, a_{OR} \rangle$	-100	-100
$\langle a_{OR}, a_{Li} \rangle$	+9	-101
$\langle a_{OR}, a_{OL} \rangle$	-100	-100
$\langle a_{OR}, a_{OR} \rangle$	+20	-50

Table 2.2 Reward for the Dec-Tiger problem.

The Dec-Tiger example comprises two agents standing in a hallway, with two doors such that behind one, there is a tiger, while behind the other, there is a treasure. The states are represented by s_l (for left) and s_r (for right), that means, $S = \{s_l, s_r\}$, indicating behind which door the tiger is hiding. The initial state distribution is uniform. Each agent can perform three actions: a_{OL} – open the left door, a_{OR} – open the right door, and a_{Li} – listen. If both agents open a door with the treasure behind it, at the same time, they get the highest joint reward $(+20)$. For an agent opening the door behind which there is a treasure, the individual return it receives is $(+10)$, and for opening the door behind which the tiger is hiding, the reward is (-100). Opening the door jointly with the tiger behind it is less severe, and the negative joint reward associated with this is (-50). A good strategy is to listen first, but then that would have a cost (or a negative reward) of (-1) for each agent. A full reward model is given in Table 2.2.

$\vec{a}$/*transition*	$s_l \to s_l$	$s_l \to s_r$	$s_r \to s_r$	$s_r \to s_l$
$\langle a_{OR}, anything\rangle$	0.5	0.5	0.5	0.5
$\langle a_{OL}, anything\rangle$	0.5	0.5	0.5	0.5
$\langle anything, a_{OL}\rangle$	0.5	0.5	0.5	0.5
$\langle anything, a_{OR}\rangle$	0.5	0.5	0.5	0.5
$\langle a_{Li}, a_{Li}\rangle$	1.0	0.0	1.0	0.0

Table 2.3 State transition function.

$\vec{a}$	s_l		s_r	
	o_{HL}	o_{HR}	o_{HL}	o_{HR}
$\langle a_{Li}, a_{Li}\rangle$	0.85	0.15	0.15	0.85
otherwise	0.5	0.5	0.5	0.5

Table 2.4 Individual observation probabilities.

Each agent can hear the tiger by means of o_{HL}, which shows that the tiger is behind the left door, or o_{HR}, shows the tiger is behind the right door. Moreover, the observation received is only 85% accurate. That means, with the probability of only $0.85 \times 0.85 = 0.7225$, the agents simultaneously receive the correct observation. The received observation is informative only if both agents listen together. Otherwise, the environment gets reset to s_l or s_r with an equal probability, and the agents receive an uninformative observation probability drawn uniformly. We note that there is *no* way an agent can observe or know that the problem has been reset. From here on, the problem continues while the hope is that agents open the door with the treasure multiple times. The transition function and observation function are shown in Table 2.3 and Table 2.4, respectively.

2.2.4 Relevant Algorithms for Decentralized Frameworks

Dealing with a problem that models multiple agents together is much harder to handle than breaking the problem down into several smaller problems and solving each one at a time, giving us the flexibility to understand each component in detail and their cohesiveness. In general, problem factoring/decomposition is a key element of MAP research [29]. Many algorithms have been proposed for solving Dec-POMDPs [46, 62, 64, 103], and many others [89, 99]. This involves breaking down a *complex* problem (e.g., Dec-POMDP [7] is shown to be NEXP hard) into multiple smaller (possibly *independent*) portions that are manageable and easily understandable. Later, individually, each is handled, yielding a distributed algorithm.

For the Qualitative Framework

In the Background section, we defined the *Qualitative* Dec-POMDP model formally and discussed the factored model and a representation of the joint policy that is also the solution to QDec-POMDPs. Next, we briefly discuss the approaches that appear in the literature to solve the QDec-POMDP model and are related to our work.

Approach 1: A Compilation-Based Approach Brafman, Shani, and Zilberstein (2013) proposed a compilation-based approach, inspired by the MPSR's translation method [16], for solving the QDec-POMDP problem. Their proposed method compiles QDec-POMDP planning to classical planning. The solution obtained for the compiled problem corresponds to an executable global plan tree, branching on agents' observations (as shown in Figure 1.1). In achieving that, their approach relies on an approximate, sound but incomplete notion of belief state [18].

The authors demonstrated overall the advantage of this compilation method over Dec-POMDP solvers using several examples. Their approach solved small problems much faster, and it scales well to larger problems compared to existing Dec-POMDP solvers.

However, this compilation-based method is designed to solve *deterministic* QDec-POMDPs, i.e., the ones in which the actions have deterministic effects. Their idea could, in principle, be expanded to handle non-determinism. We can achieve this by embedding the uncertainty of the action effects into the uncertainty of the initial belief [106], but this will affect the solution size and the time it takes to solve a problem.

Approach 2: Iterative MAP A major drawback of the compilation-based approach is that the compiled problem becomes too large for an off-the-shelf classical planner; as a result, its scalability is limited. As we know, one key difficulty in distributed execution is the need to construct a joint policy. The problem with this compilation approach was that it performed branching on the combinations of joint observations.

Bazinin and Shani (2018) proposed an interactive process, called Iterative MAP (IMAP [5]), which exploited the fact that *interactions* between the agents in cooperative settings are often limited in real-world scenarios. They proposed to solve as much as possible of the MAP problem locally, provided that the overhead of ensuring that the local policies of the agents are properly coordinated is not too large, which can be expected in such MAP problems. At least abstractly, IMAP is reminiscent of the iterated best-response method used by the JESP algorithm for Dec-POMDPs [56]. (JESP is discussed next.)

At a very high-level, Iterative MAP works as follows. It focuses on one agent at a time. An agent solves some part of the MAP problem and generates constraints for other agents. The “next” agent will try to augment its solution such that the constraints imposed on it are satisfied. Moreover, this agent also generates constraints for other agents. If the constraints imposed on it are not satisfied, the process backtracks, and the first agent replans. Thus, the IMAP algorithm solves multiple single-agent problems until all agents’ plan trees are coordinated.

For deterministic QDec-POMDPs, IMAP showed better performance than the compilation-based approach used in [18] and much better results than the Dec-POMDP solvers [5]. IMAP showed great promise in scaling up to larger QDec-POMDPs.

For the Quantitative Framework

Since solving Dec-POMDPs to find exact solutions is hard, compact representations [62, 103] are proposed for solving these models efficiently. The main idea is to consider an explicit form for representing transitions, rewards, states, and observations, all comprising several factors [64], and to have an explicit model that carefully expresses how these components interact with and affect each other [63].

Joint Equilibrium based Search for Policies The *best response* is roughly defined as finding the best policy for an agent and keeping the policies of other agents fixed . This is applied to a large subset of games like congestion games [76] and potential games [54], where computing the iterated best response leads to equilibrium. Researchers have used dynamic programming for computing the best response for solving Dec-POMDPs [56], too.

Nair *et al.* (2003) addressed a method to reduce the complexity of generating a joint policy by focusing on one agent at a time instead of building a joint policy while considering all agents at once. *Joint equilibrium* based *search* for *policies* (JESP) iterates over all the agents, one at a time, and finds an optimal policy while keeping the other agents’ policies fixed. It generates the best response for this agent based on dynamic programming. However, it continues iterating over the agents until no improvements in the joint reward are received.

As a result, the idea achieves local optimal Nash Equilibrium, called JESP. However, one can easily extend the results to stochastic games like POSGs [64].

JESP scales up the Dec-POMDP model by an order [56], but overall it has shown only limited scalability as the complexity associated with considering the requirement of joint-observations does not reduce, as also pointed out by Bazinin and Shani (2018) in [5]. Perhaps in a general sense, the IMAP algorithm is an implicative form of the iterative best response approach used in the JESP [56].

Chapter 3

A Factored Approach to Multi-Agent Planning with Partial Observability

Publication(s)

1. Shashank Shekhar and Ronen I. Brafman, and Guy Shani, A Factored Approach to Deterministic Contingent Multi-Agent Planning, in Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July 11-15, 2019, pages 419-427.

3.1 Motivation and Overview

Many automated planners proposed to use the loose interactions between the components of a planning problem to solve it efficiently. For example, hierarchical planning approaches decompose a planning domain into loosely interacting parts (sub-domains). It helps decompose the goal into multiple sub-goals that are high-level operators. The planning approach works level by level, independently, such that the obtained sub-plans are combined carefully to build the final plan. The idea of domain decomposition is not limited to hierarchical planning, as planners like in [47] also use this idea. Perhaps they do not need an explicit hierarchical structure, but they often need to backtrack over sub-domains and end up replanning for the sub-goals they have already achieved. In general, this approach is inefficient as the need to backtrack increases the overall complexity, and it often dominates the forward search complexity.

Amir and Engelhardt [2] proposed to exploit the structure of a planning problem by an approach that dynamically factors the planning problem. They give a generic planning method that exploits it to generate a plan faster, which is sound and complete. They show that their idea that is composed of “factoring” and “planning”, scales well to large domains. Later, researchers also proposed to utilize the domain structure to make the centralized search faster (e.g., see [14, 30, 31]). Further, Brafman and Domshlak (2008) showed that their approach can also exploit the “multi-agent” (“distributed”) part of MA-STRIPS problems [14]. Note that, in these methods, a centralized entity would have complete access to the planning problem.

Therefore, it is quite natural to seek a problem factoring approach for “non-classical” planning. To be precise, we are interested in the MA planning problem that models uncertainty and partial observability. The nature of a non-classical planning domain is different from the nature of a classical planning domain, and for such MAP problems, non-sequential solutions (i.e., often in the form of a tree or graph) are generated. We present a new approach for solving *deterministic qualitative* Dec-POMDPs, mainly motivated by the ideas behind factored algorithms for classical planning, in which factoring is achieved by defining the problem as a MAP problem [14]. Our contributions in this chapter are as follows. Based on [84], we first describe a problem factoring approach along with its theoretical guarantees. In the empirical evaluation section, we introduce a new MAP domain, called ***h**eavy-**s**tructural **d**amage* (HSD). In total, we perform experiments in three domains (including HSD). The experiments show that the approach scales to large problems compared to the contemporary QDec-POMDP and Dec-POMDP solvers.

3.2 Introduction

This chapter describes an approach, called QDec-FP, motivated by problem factoring for solving QDec-POMDPs in which single-agent problems are solved repeatedly. Not only does this approach show superior empirical performance, but it is also cleaner algorithmically and hence comes with simple theoretical guarantees. This approach has great potential for concurrency – allowing for truly distributed computation by a group of agents, and is likely to be extendable to offer privacy-preserving properties, as well.

At a high-level, the QDec-FP approach works as follows: First, we solve a MAP problem in which we assume that communication is free and immediate. Hence, all agents "see" each others' observations. This is a *single-agent* planning problem in which the actions available are the union of all agents' actions, and the observations are the union of all agent's observation actions. Thus, while this problem is not as simple as that of generating a policy for a single agent, because we have a larger action and observation space, it is not a MA planning problem: we need to track only a single belief state, and we do not need to reason about multiple concurrent belief states and to coordinate between agents with different states of information.[1] We refer to this as the *team planning problem.*

From the policy tree obtained by solving the team problem, we extract for each agent a sub-tree that contains only the actions of that agent that impact other agents. These could be collaborative actions (i.e., actions that require joint concurrent execution by multiple agents, such as lifting a table or pushing a heavy box [82]), or actions that supply preconditions to other agents' actions. We refer to this as the agent's projection of the policy tree.

Agent φ_i's projection is not likely to be executable by it for two possible reasons. First, it may require φ_i to perform action a only when p holds, where p was observed before in the original team policy by another agent φ_j. In the team problem, φ_i learns the results of such observations immediately and for free, but in the real domain, φ_i must somehow obtain this information. Second, the projected solution removes all φ_i's action that are not needed by other agents. Some of these removed actions may have supplied some precondition to one of the remaining actions. Thus, the next step in the algorithm is to let each agent turn its projected policy into an executable policy. To do this, each agent solves a local planning problem in which its goal is to perform all the actions in its projection under the same conditions. Thus, its solution would be a policy that the agent can execute (provided the other agents execute their actions in the team policy).

Of course, the single-agent problems may not be all solvable, in which case we must backtrack and seek a new team solution. But if they are all solvable, then we can get a

[1] For flat models, single-agent contingent planning is in PSPACE [49] and MA contingent planning is NEXP-TIME hard [7].

legal joint policy for the original problem by taking the solutions of each projected problem and aligning its actions properly so that collaborative actions are executed at the same time, and preconditions are supplied before their consuming actions. Thus, if the underlying single-agent contingent planner is able to generate all solutions (e.g., by using AO^* as the underlying search algorithm), then our method is complete.

We implemented and tested this approach using a single-agent, an off-the-shelf contingent solver, CPOR [43], which generates a policy tree by repeatedly calling the online contingent planner SDR [81]. SDR generates a single execution branch that corresponds to the true initial state. CPOR simply calls it multiple times with different "true" initial states. We compare our factored planning based with IMAP on the two domains described in that paper, and on a new disaster support domain. Our factored planning algorithm scales better both as the number of objects in the domain increase and as the number of agents increases. In addition, it typically generated smaller, and more balanced policy trees.

3.3 A Factored Approach to Solving QDec-POMDPs

We now present a quite general scheme for factored planning in QDec-POMDPs. In fact, in principle, with suitable modifications and a suitable single-agent contingent planner, this approach works for non-deterministic domains, as well.

In Chapter 2, we gave the basic definition of a flat-space QDec-POMDP [18], its solution (*i.e.*, a set of individual policy trees, one for each agent or a joint-policy tree), and a factored definition motivated by contingent planning model definitions [11]. Definition 4 represents flat-space *Qualitative Decentralized* Markov Decision Processes. After that we formally describe its solution, *i.e.* a joint policy tree that bases/extends/uses the single-agent contingent policy representation. Note that the single-agent *contingent policy* is formally defined in Definition 2 (Contingent Solution).

The high-level structure of the QDec-FP algorithm is described in Algorithm 1. First, we generate the single-agent *team problem*, which is a single-agent contingent planning problem. This is simply obtained by taking the original multi-agent problem, treating all actions as if they are executed by a single agent and all observations are observed by this “hypothetical” single-agent that controls all the agent together. The solution obtained for this modified problem is called the *team solution* tree and denoted as τ_{team}. Next, τ_{team} is projected to each agent, obtaining τ_i for $i = 1,\dots m$. Now, each agent solves a planning problem whose goal is to generate an executable local policy that contains τ_i as a projected sub-tree. That is, each action in τ_i is executed in this local policy under the same conditions and in the same order. If all problems are solvable then we align the actions in their solution and return a solution.

If one of the agents cannot solve its local problem, we generate a new team solution and repeat the process. If no new team solution remains, we fail.

Algorithm 1 Factored Planning for QDec-POMDP

```
 1: procedure QDEC-FP($\langle I, P, \vec{A}, Pre, Eff, Obs, b_0, G\rangle$, output $\{\tau_i\}_{i=1}^{|I|}$)
 2:     Set $P_{team} = \langle P,\ \cup_{i=1}^{m} A_i,\ b_0,\ G\rangle$
 3:     while (unexplored solutions to $P_{team}$ exist) do
 4:         $\tau_{team}$ = Contingent-Solve-Next($P_{team}$)
 5:         foreach agent $\varphi_i$ do
 6:             $\tau_i$ = Project($\tau_{team}, \varphi_i$)                ▷ Algorithm 2
 7:             $\tau_i^*$ = Compress($\tau_i$)                           ▷ Algorithm 3
 8:             $P_i$ = Generate-Contingent($\tau_i^*$)
 9:             $\tau_i'$ = Contingent-Solve($P_i$)
10:             if unsolvable $P_i$ then
11:                 GOTO 4
12:             end if
13:         end foreach
14:         $(\tau_1'', \ldots, \tau_m'')$ = Align($\tau_1', \ldots, \tau_m'$)
15:         return $(\tau_1'', \ldots, \tau_m'')$
16:     end while
17:     return failure
18: end procedure
```

3.3.1 The Team Problem

Generating the team problem is easy. We take the original QDec-POMDP and simply treat all agents as objects under the control of some super-agent. This super-agent also receives all the observations. This team problem is now a single-agent contingent planning domain.

3.3.2 The Single-Agent Problems

Once we have a solution τ_{team} to the team problem, we generate one projection τ_i of it for every agent (see Algorithm 2). The projection is obtained by removing from the tree all non-observation actions except those executed by agent φ_i. Among the actions of agent φ_i, we leave only actions that impact other agents directly. An action impacts another agent directly if either (1) it is a collaborative action; (2) it supplies a precondition to an action of another agent; or (3) it achieves a goal proposition. In factored planning such actions are often referred to as *public* actions, while the remaining actions are called *private* [14]. Similarly, a proposition that appears in the description of multiple agents is called *public*,

Algorithm 2 A high-level algorithm for Project(τ_{team}, φ_i): Generates a projected plan tree τ_i for the agent φ_i provided the team plan τ_{team}. To keep this procedure clean, we skip the optimizations used in our implementation along with other minor details.

```
relevantAS = relevant action set containing all the public actions of $\varphi_i$, and for simplicity,
    we assume that it also contains all the observation actions (irrespective of the agents they
    belong to) appearing in $\tau_{team}$
procedure Projection($\langle \tau_{team}, \varphi_i \rangle$, output ref $\{\tau_i\}$)   ▷ create a copy of $\tau_i$ in each call
    if root($\tau_{team}$).action then
        while TRUE do
            if root($\tau_{team}$).action $\in$ relevantAS then
                $\tau_i$.action = root($\tau_{team}$).action
                break
            end if
            $\tau_{team} = \tau_{team}$.single-child
        end while
    end if
    if root($\tau_{team}$).single-child then   ▷ if $Obs(N.action) \neq \emptyset$, $N.single\text{-}child$ is NULL
        while TRUE do
            act = root($\tau_{team}$).single-child.action
            if act $\in$ relevantAS then
                $\tau_i$.single-child = Projection(...)   ▷ with $\tau_{team}$.single-child as input
                break
            end if
            $\tau_{team} = \tau_{team}$.single-child
        end while
    end if
    if root($\tau_{team}$).left-child then
        while TRUE do
            act = root($\tau_{team}$).left-child.action
            if act $\in$ relevantAS then
                $\tau_i$.left-child = Projection(...)   ▷ with $\tau_{team}$.left-child as input
                break
            end if
            $\tau_{team} = \tau_{team}$.single-child
        end while
    end if
    if root($\tau_{team}$).right-child then
        while TRUE do
            act = root($\tau_{team}$).right-child.action
            if act $\in$ relevantAS then
                $\tau_i$.right-child = Projection(...)   ▷ with $\tau_{team}$.right-child as input
                break
            end if
            $\tau_{team} = \tau_{team}$.single-child
        end while
    end if
    return $\tau_i$
end procedure
```

Algorithm 3 A high-level algorithm for Compress(τ_i): Generates a compressed plan tree τ_i^* from a projected tree τ_i. For simplicity we skip the optimizations (and other minor details) used in the implementation.

```
procedure COMPRESSION($\langle\tau_i\rangle$, output ref $\{\tau_i^*\}$)    ▷ create a copy of $\tau_i^*$ in each call
    if root($\tau_i$).action then
        $\tau_i^*$.action = root($\tau_i$).action
    end if
    if root($\tau_i$).single-child then
        $\tau_i^*$.single-child = COMPRESSION(...)    ▷ with $\tau_i$.single-child as input
    end if
    leftC = $\tau_i$.left-child and rightC = $\tau_i$.right-child
    if leftC or rightC then
        if COMPARE(leftC, rightC) then
            $\tau_i^*$ = COMPRESSION(...)    ▷ with $\tau_i$.right-child as input
        end if
        if not COMPARE(leftC, rightC) then
            $\tau_i^*$.left-child = COMPRESSION(...)    ▷ with $\tau_i$.left-child as input
            $\tau_i^*$.right-child = COMPRESSION(...)    ▷ with $\tau_i$.right-child as input
        end if
    end if
    return $\tau_i^*$
end procedure
```

and a proposition that appears in the description of only a single agent's actions is called *private*. For example, ignoring collaborative actions for the moment, in the Box Pushing example, the *push* actions of different agents are public, assuming multiple agents can push the same box. The *move* actions, however are private. Similarly the location of a box, which can be influenced by multiple agents is public, while the location of an agent, that can be influenced only by its *move* actions, is private.

Collaborative actions are a bit more subtle. A *joint-push* action is performed by multiple-agents. We treat it as a public action. However, when considering whether a proposition is private or not, we consider only its part of the action. Thus, *joint-push* requires both agents to be located in the same position, making an agent's location appear public. However, since we will ensure that joint-actions are respected by all participating agents, the other agents need not be concerned with the preconditions of these actions that are otherwise private to other agents. Thus, one agent need not know or care about the location of the other agent – this is the latter agent's problem – as long as that agent executed the collaborative action at the same time.

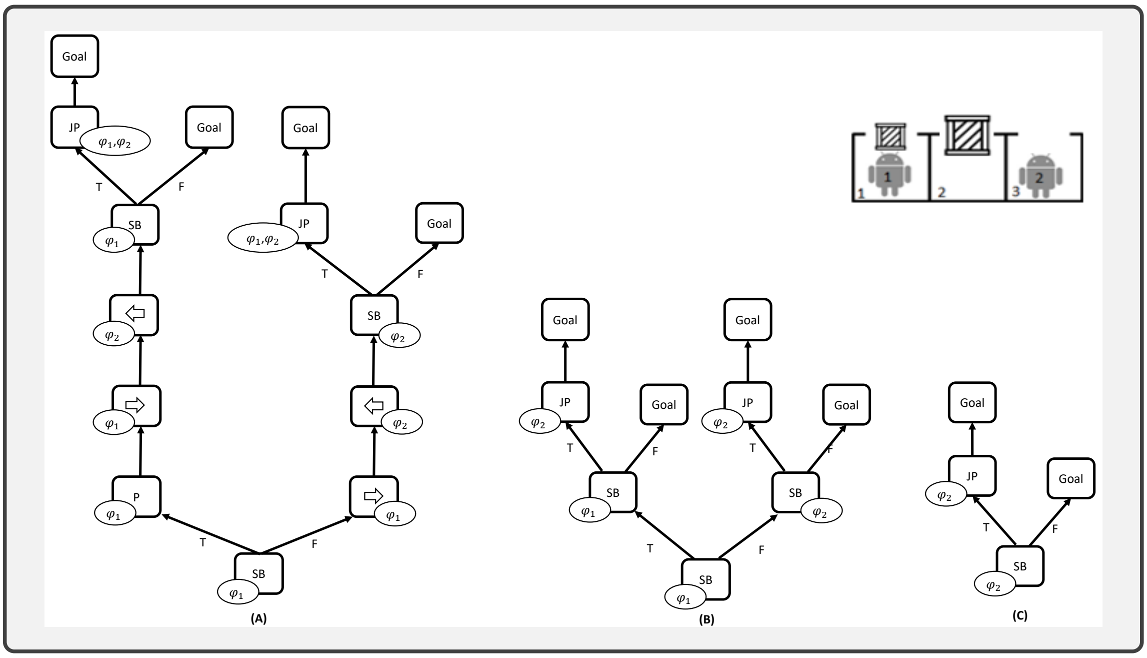

Fig. 3.1 The details of this figure are as follows. (A) A team solution plan for a problem with two agents, φ_1 and φ_2, a light box and a heavy box that need to be outside the grid in the goal state. (B) Its projection to φ_2. Notice that observations include those ofφ_1, too. (C) Compacted projection – no sensing is required by φ_1.

Thus, each projection τ_i is a tree containing observation actions, possibly by other agents, and the public actions of agent φ_i. Furthermore, if a is an action in τ_i, we remove from it any public precondition, that is, one supplied by another agent in τ_{team}.

Next, we must ensure that all observations are necessary. Consider, for example, a case in which τ_i includes sensing the value of p, but the agent acts identically whether p is *true* or *false*. The reason the observation for p exists in the original team solution τ_{team} is probably because some other agent needs to differentiate between these two cases. If we leave this distinction in place, we risk losing completeness if our agent is unable to observe p.

To remove redundant observations, we apply standard graph algorithms: Moving bottom-up, whenever the two sub-trees below an observation node are identical, we remove the observation and retain just one of the sub-trees (see Algorithm 3). When comparing sub-trees, two sensing actions by different agents that sense the same proposition are treated as identical, and only the “single-agent” element of the collaborative actions is considered – i.e., we do not distinguish between two lift-table actions in which the other agents collaborating with φ_i are different, because the action φ_i executes in this collaboration is the same. An example of a team solution, its projection, and the compacted projection is given in Figure 3.1.

The projected tree is typically not executable by the agent. It contains observation actions that are not its own, and some actions are not supplied with their preconditions by previous actions. Our goal is to extend this tree, by replacing the observation actions of other agents by the agent's observation actions, when relevant, and adding private actions that supply missing preconditions. Note that only private actions are added – if an agent adds a public action that requires that another agent will supply it with a precondition, or one that changes the value of a public proposition, this might impact the other agents, either requiring them to modify their plan, or destroying a precondition they need.

Under the assumptions that all other agents execute their public actions in their projection, this tree is executable because all actions have the preconditions supplied either by other agents or by the agent. The resulting policy tree, τ_i^{sol}, should have the property that, when projected to contain only observations and public actions of φ_i, it is identical to τ_i, with the exception that observations of other agents are replaced by those of φ_i.

Thus, the next step is to take the compact policy tree obtained (which is also denoted τ_i) and generate a single-agent contingent planning problem. The goal of solving this problem is to generate an executable policy tree that contains all the actions in the projected tree, where these actions are executed under the same conditions and in the same order.

The single-agent planning domain is generated as follows:

1. The set of actions contains all grounded actions appearing in τ_i and all private actions of φ_i.

2. The preconditions that other agents achieved in τ_{team} are removed

3. Each action in a leaf node has the added effect *done*; in case of a branch in which execution terminates after an observation, a dummy action that achieves *done* is appended.

4. Each non-leaf action a has a special added effect p_a

5. Each non-root action a has an added precondition $p_{a'}$, where a' is the parent of a in the tree

6. The first action in a branch following an observation has the appropriate value of the observation as an added precondition. If there are multiple consecutive observations without intermediate actions, then the value of all of them in this branch have to be in its preconditions.

7. Initially, all added propositions are *false*

8. The goal of the planning problem is *done*.

Example 2 *Consider the projected tree in Figure 3.1(C). For this tree, we generate a planning domain for φ_2 with the following actions:*

- *The original description of φ_2's private actions – in this domain these are the various movements of the agent.*
- *The action SenseBox$_{2,\varphi_2}$ (i.e., SB$_{\varphi_2}$) with an added effect: observed$_{box2,\varphi_2}$*
- *The action Dummy-Out with preconditions SensedBox$_{2,\varphi_2}$ and $\neg$BoxAt$_2$ (the observed value). Its only effect is done.*
- *The action $JointPush_2$ (i.e., JP). Its preconditions contain φ_2's private preconditions, i.e., AgentAt$_{\varphi_2,2}$ and the observation value for this branch: BoxAt$_2$. Its effects are the original effects and done.*

The added propositions: SensedBox$_{2,\varphi_2}$ and done are initially false.

3.3.3 Alignment

If all projected problems are solvable, we still need to align them to ensure that the joint-policy is executable online. This is done by concurrently going over all trees level by level and ensuring that all actions are executable and that all collaborative actions are executed at the same step. If an action is not executable at the current step, or if only one part of a collaborative action is scheduled to a time-step, a *no-op* is added to the relevant branch so that the execution of the action is delayed to the next step.

3.4 Soundness and Completeness

Theorem 1 *The QDec-FP algorithm is sound if the underlying single-agent contingent planner is sound.*

Proof First, we observe that the joint policy contains all public actions that appear in the team solution. If some action is missing then we cannot achieve *done* in the branches that contain it. Because we add branch conditions as preconditions, only execution of branch actions in sequence can make it *true*.

Next, we observe that the solution plan is executable. All preconditions supplied originally by other agents are supplied by them by the observation above. All other preconditions have to be supplied by the agent itself to obtain a valid plan, and soundness of the single-agent planner implies its validity. Of course, other agents' actions that supply an agent's preconditions must be scheduled before and cannot be destroyed. Let k be the maximal number of actions inserted between any two public actions of any branch in any agent's plan. Consider the original team plan, but now with k no-ops inserted between any two consecutive actions. This plan remains a valid team plan. Now, we have enough time steps to insert all the additional private plans in between these actions without impacting the relative order of public actions. Note that this also ensures that collaborative actions are executed at the same time. Since no new public actions can be added when solving the projected problems, one agent's solution cannot introduce actions that might delete a precondition. Hence the solution is executable.

Finally, since all goal achieving actions are public, then by the above they will be executed and the goal will be achieved in all branches, hence this policy is a solution.

Theorem 2 *The QDec-FP algorithm is complete provided the single-agent contingent planning algorithm can exhaustively generate all possible team policies.*

Proof Suppose that the multi-agent planning algorithm has a solution. It is also a solution to the single-agent algorithm, and hence it will be generated by it at some point. Its projection will contain all the public actions and observations that the agent executes in the solution. Since there is a solution, the local projections are solvable, and their solution contains all needed additional private actions. As explained in the soundness proof, there is a simple, less efficient variant of the alignment algorithm that is guaranteed to succeed if there is a team solution.

3.5 Trade-offs for Efficiency

A top level that exhaustively searches the space of team policies is unlikely to work in practice. To make the QDec-FP algorithm more efficient, we have made a few compromises. We explain them and their implications for completeness.

3.5.1 Backtracking

Algorithm 1 requires that solutions be generated one after another. In principle, this is easy to do with a systematic tree-generation algorithm such as AO^*. However, currently CPOR [43] does not support AO^*-based search, and we suspect that an AO^* planner for contingent planning is likely to be inefficient due to the lack of good heuristics for such problems. An alternative is to augment the planning domain to reflect learned no-goods so that previously generated solutions are no longer solutions for the new domain. As this only causes us to prune inappropriate team solutions, this does not affect the algorithm's completeness.

The fundamental problem is to change the domain of a single-agent contingent planner so that a previous solution will not be generated again. Consider team solution τ which was not extendable to a joint plan. We want to make sure that no future solution will be τ, or τ with some of its branches extended with new actions.

This requires that at least one branch of τ will not appear as a branch prefix in a following solution. Given a *specific* branch, we can ensure that it is not part of future solutions by adding suitable preconditions and effects to the actions in this branch, so that if they are executed in sequence, we reach a dead-end. We can then force the planner to select a specific branch of τ by requiring the first action in the plan to be one of a set of actions, each of which essentially selects one branch. However, this approach is not scalable, as this modification is required following each backtrack.

For efficiency, we actually add a weaker, but simpler constraint (albeit, ones that are still not very scalable). The main source of failure on the projected problem is the need to branch on some condition p that is not observable by the agent, followed by sub-trees that are different depending on the reason for failure. Hence, our first step is to traverse the projected sub-tree and find such branch points.

Suppose we have an asymmetric sub-tree rooted in an observation of p. Suppose that a is an action that appears on one branch following the observation, but not on the other branch. We would like to add a constraint that forces a to appear on both branches, or not to appear at all. We add a special action *commit-p* that can only appear before the observation of p. This action has a special effect that is add as a precondition to a. Thus, if p was observed, a cannot be executed unless, earlier, we executed *commit-p*. This action commits to performing a on all branches following the observation: it has an effect that negates a goal proposition, and this effect can be negated by a only.

3.5.2 Signalling

Consider a problem in which agent φ_1 can observe p and agent φ_2 can observe q, but not p, and that the solution requires that agent φ_2 will act differently depending on p's value. In general, the problem is unsolvable. However, suppose that φ_1 can control the value of q. It can then signal to φ_2 the value of p by manipulating q. In theory, if we generate every possibly team policy, we can generate such a policy as well. There is a caveat, though. The team solution will branch on p, φ_1 will align the value of p with q and then it will branch on q. For our algorithm to remove branching on p in φ_2's projection, the two sub-trees that correspond to the two possible values of p must be identical. A decent single-agent contingent planner will not insert a redundant observation – and since observing q adds no value in this case, it is redundant. Furthermore, even if it does, the branches given $p \wedge \neg q$ and $\neg p \wedge q$ will be left empty, as they never occur.

One ad-hoc way of addressing this is using signalling procedures to replace observations. That is, if the projected problem for φ_2 requires sensing p, we can try to replace it in the plan by some signalling sub-routine/macro followed by an observation of the signal. This undermines the separation between the team solution and local solutions, since signalling involves two agents. However, it is likely to be practical. Another option is to try to keep track of the belief states of agents during the team planning. To some extent, this can be done syntactically, and was done in earlier work on compiling conformant and contingent planning into classical planning [65]. However, this approach does not scale too well. We return to this issue of signaling in the next Chapter.

3.6 Empirical Evaluation

In this section, we provide an empirical analysis that shows that our QDec-FP approach scales much better than IMAP. The IMAP paper considers two domains: Box-Pushing (BP) and Rovers, and we add a novel domain called Heavy-Structural-Damage (HSD) domain.

3.6.1 MAP Domains

We briefly describe all three domains below.

Box-Pushing (BP)

There are boxes situated in a grid-like structure. Each box is supposed to be moved to its destination location, in this case at the edge of the grid, i.e., at the end of the column the box appears in. Each box is either at some location in the grid or at the goal location. An agent needs to be in the same grid cell where a box exists to observe it and to push it. Boxes are either heavy or light. Two agents are required to successfully *push* a heavy box, while a single agent can push a light box. Agents can also move between two adjacent locations in the four primary directions. In this domain, we model uncertainty in terms of the initial locations of the boxes. The agents are non-homogenous — different agents can observe and push different boxes.

Heavy-Structural-Damage (HSD)

This domain captures a scenario where in a grid like locality, due to an earthquake, several buildings have been collapsed. Debris is scattered all around and there is the possibility that underneath some pillar/beam of a collapsed building there is a victim. Agents need to search the victims and rescue them. An agent can *sense* a patient. If the patient is underneath a pillar or a roof beam, the patient will be rescued to a safe location. If the patient is an adult a joint rescue operation will be performed by more than one agent. Agents can move to connected locations in the maze. The goal is to rescue all the victims.

Rovers

Multiple rovers navigate a planet surface, finding samples and communicating them back to a lander [5]. Two rovers must simultaneously collect the rock sample, while a single agent can sample soil as well as take images of certain objectives on its own. Coordination points include locations (waypoints) which are accessible to multiple rovers. Rovers communicate sampled soil/image/rock data to the lander that exists at a certain waypoint. A rover navigates between two waypoints and must be present at the corresponding waypoint to sample. Availability of data to sample at a waypoint is unknown to the rovers initially. In this modified domain, our schema requires two rovers working jointly to collect rock samples. After taking measurements, the rovers must broadcast them back to the lander. In this domain in general, a rover has fewer public actions (approx. 46% on average), but a relatively complex internal planning problem, including navigation, soil and rock sampling, and image capturing.

3.6.2 Experiments

We compare our algorithm called the QDec-FP planner (based on factored planning) with the *Iterative* MAP (IMAP). Both planners were run on a Windows 10, 64-bit machine with $i7$ processor, 2.8GHz CPU, and 16Gb RAM. Both QDec-FP and IMAP are implemented in C#. For IMAP, we used the code by Bazinin and Shani (2018) [5].

The results are shown in Table 3.1. We describe the running time, and size of the resulting plan tree which is measured in terms of the number of branches and the height. The number of branches is also indicative of the number of sensing actions performed, as branching occurs following an observation. We depict only the maximum values of width and height of individual plan trees obtained for all the agents. Table 3.1 shows that the factored planning approach scales better than IMAP. Specifically, the increase in the number of objects had minor impact on running time, as opposed to IMAP. Even more markedly, increasing the

Domain	Ins (#agt)	Size	QDec-FP *vs* IMAP					
			Max Width		*Max Height*		*Time (sec)*	
Box-Pushing	P01 (3)	12	8	8	10	11	1.7	8.7
	P02 (3)	15	16	16	26	15	4.6	18.8
	P03 (4)	16	4	8	5	17	1.4	44.6
	P04 (5)	19	4	32	4	15	1.6	97.5
	P05 (5)	21	38	-	20	-	13.6	-
	P06 (6)	25	4	128	9	31	2.7	130.2
	P07 (9)	36	64	-	24	-	25.3	-
	P08 (10)	37	90	-	29	-	41.1	-
	P09 (12)	46	64	-	23	-	33.9	-
	P10 (12)	46	128	-	24	-	43.8	-
	P11 (12)	48	128	-	26	-	60.2	-
HSD	P01 (3)	14	8	8	11	9	1.2	7.1
	P02 (3)	14	6	8	15	12	2.4	8.6
	P03 (4)	20	8	20	9	14	5.1	70.7
	P04 (6)	32	4	64	7	29	3.2	208.9
	P05 (7)	36	4	128	7	33	3.8	402.8
	P06 (7)	36	112	-	31	-	35.6	-
	P07 (8)	40	110	-	29	-	53.8	-
	P08 (8)	42	148	-	36	-	81.5	-
	P09 (9)	47	128	-	31	-	88.3	-
	P10 (9)	47	127	-	34	-	129.5	-
Rovers	P01 (1)	12	2	2	9	9	0.5	0.4
	P02 (2)	14	2	2	8	8	0.7	0.8
	P03 (1)	12	4	4	15	15	0.8	1.5
	P04 (2)	17	11	12	21	43	3.1	20.8
	P05 (2)	17	12	-	24	-	3.5	-
	P06 (2)	17	27	27	43	52	7.2	267.6
	P07 (2)	28	35	-	35	-	8.5	-
	P08 (2)	28	31	-	37	-	7.6	-

Table 3.1 Performance comparison of our QDec-FP planner and the IMAP approach. *Ins* is instance number with the number of acting agents in the brackets. *Size* denotes the number of objects considered in each problem. *Maximum Width* and *Maximum Height* respectively show the values of maximum number of branches and maximum height of all individual solution trees obtained for the agents. Time is in seconds.

number of agents had a much smaller influence on the running time of the factored planning algorithm. Thus, except for the smallest problem, the factored planning algorithm is faster than IMAP and scales to much larger problems. And, except for only a few problems, the policy trees generated by the factored planning algorithm are smaller. In fact, upon examining the policies generated, we observe that the factored planning approach often generates more balanced trees for different agents, while IMAP generates trees with significantly different sizes. Finally, in the HSD domain, we can also see that increasing the number of agents does not have a major impact on run-time of the factored planning approach.

In most of the tested domains, the agents are homogenous – they have the same actions. In these domains, backtracking is not needed. The instances P02 and P05 in the Box-Pushing domain, and the instances P02 and P03 in the HSD domain contain non-homogenous agents. This causes backtracking to occur in all of these problems because when an agent is asked to act differently following a sensing action's results, and the agent cannot perform this sensing action, then agent roles must be changed. We can see that in Box-Pushing P05, the need to backtrack caused the planner to take more time than a slightly larger problem that does not require backtracking (P06). Similarly, in HSD, P03 takes more time than the slightly larger P04. Nevertheless, the running times were still quite reasonable and in all these instances, the factored planner did much better than IMAP.

As we mentioned earlier, the IMAP approach was compared to Dec-POMDP algorithms on the BP domain, and scaled much better. Thus, one can reasonably conclude that, for the special class of deterministic Dec-POMDPs where the uncertainty is only w.r.t. the initial state, our factored approach is able to handle much larger instances than Dec-POMDP algorithms.

3.7 QDec-FP: A Brief Summary

This chapter described a factored approach for solving multi-agent contingent planning problems. The MA problems were modeled using the *qualitative* Dec-POMDP model, and the solution approach works by taking a team solution – i.e., one in which all agents have immediate access to every other agent's observations – and fixing it so that it becomes executable online by all agents. In our experimental evaluation, we compared the factored planning algorithm with IMAP, a recent algorithm that, at least on deterministic problems, scales to much larger domains than current algorithms for Dec-POMDPs. The factored planning algorithm is almost always faster, and often much faster than IMAP, scales better with increased agent numbers, and typically generates smaller policy trees.

Problem factoring is a key element in MAP in general (e.g., see [29]) and in many algorithms for Dec-POMDPs (e.g., [46, 62, 64, 103]. It is particularly natural to factor the problem among the different agents, yielding a distributed algorithm. In particular, we believe it is natural to study algorithms in which the information exchanged by agents centers on their *commitments* to other agents. This is essentially what our algorithm does – each agent tries to plan to fulfill its commitments in the team plan. While commitments bring to mind the idea of *influences*, studied in the Dec-POMDP literature [64, 103], they are different. Influences are used to separate the belief state of one agent from all other information – they capture the variables that directly influence the agent. Distributions representing variables external to the agent can be marginalized to these influences, reducing the state space that an agent needs to consider, e.g., when computing the best response. Commitments can be viewed as an analogous concept at the level of the policy – they refer to the actions that an agent performs to facilitate the actions of another agent.

The factored planning approach has a number of advantages: it is conceptually simple, and hence easy to analyze theoretically; the second stage of the algorithm generates independent sub-problems for all the agents and can be parallelized; and since each agent solves a local planning problem, a privacy-preserving variant is likely to be formulated. Moreover, efficient backtracking and signaling are still quite challenging, and we also did not test our methods on non-deterministic domains. On the other hand, a method that tries to generate a MA policy directly, instead of first committing to a team plan, may lead to better solutions.

Chapter 4

Improved Modeling of Agent's Knowledge

Publication(s)

1. Shashank Shekhar and Ronen I. Brafman, and Guy Shani, Improved Knowledge Modeling and Its Use for Signaling in Multi-Agent Planning with Partial Observability, in Proceedings of the 35th Association for the Advancement of Artificial Intelligence (AAAI) 2021.

2. Shashank Shekhar and Ronen I. Brafman, and Guy Shani, Signaling in Contingent Multi-Agent Planning, in (*online*) Proceedings of the Workshop on Epistemic Planning (EpiP'20), ICAPS 2020.

4.1 Motivation and Overview

The QDec-FP approach described in Chapter 3 suffers from a few shortcomings that stem related to the way team planning works. Team planning considers the MAP problem as a single-agent contingent problem, models all agents together as if they are one agent with access to all observations. As a result, team planning may generate a team solution that is less likely to get extended to a sound joint-plan, as all the agents may not be able to fix their projected parts, causing QDec-FP to backtrack.

In Chapter 3, we briefly discussed *signaling*, where an agent φ_i can sense variable p and agent φ_j cannot, but it can sense another variable q. φ_i can control q, in this case φ_i can signal p to φ_j by carefully *setting* (e.g., by changing the state of the world in a predefined way) $p \equiv q$. However, currently, there is no reason for the solver to introduce signaling at the level of team planning since once φ_i senses p, φ_j would know its value immediately as if φ_j had sensed its value. That is, the different knowledge each agent has is not captured by the team problem. Therefore, the QDec-FP planner cannot handle MAP problems that need signaling to be solved.

Our contributions in this chapter are as follows. In the next section, we give a detailed explanation of the drawbacks associated with the QDec-FP approach. We then describe a new planner that uses richer information (to overcome the effects of those drawbacks) about agents' knowledge. It also enables us to model *signaling*, which is covered next, and then we discuss the planner's theoretical properties. In the empirical evaluation section, we introduce a new MAP domain with components to capture partial observability. In the experiments performed on *three* MAP domains (including two existing and a new one), we compare our new planner, called QDec-FPS, and the QDec-FP planner. We then discuss the applicability of the new approach. This approach got published in 2021 [86], while a partially complete work got accepted in 2020 [85].

4.2 Drawbacks of QDec-FP's Team planning

An example of a team solution, its projection, and the compacted version is shown in Figure 4.1. Although a similar example is given in Chapter 3, here we use this example specifically for pointing out the drawbacks of team planning.

In Figure 4.1(A), we can see the team solution, containing both private and public actions of the two agents. The team plan assumes shared knowledge. We can see that in the left (or right) branch, only agent φ_1 senses the heavy box, and both the agents immediately know about the box, and then they jointly push it if needed. Figure 4.1(B) shows the projected tree

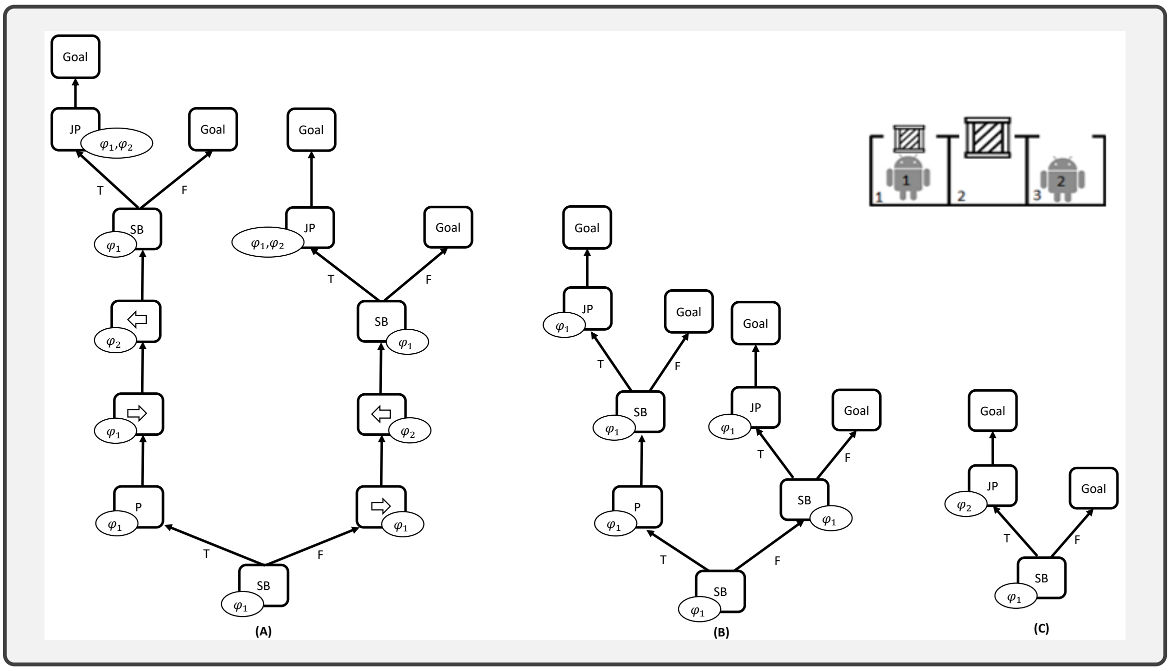

Fig. 4.1 (A) Team plan tree τ_{team} for a problem with two agents, a light box and a heavy box that need to be outside at the edge of the grid in the goal state. (B) The projection of τ_{team} to φ_1 for which all the sensing actions of τ_{team} are non-redundant and remain. (C) A compacted projection for φ_2 in which no sensing action by φ_1 is required.

of φ_1. All the private actions were removed, while all the sensing actions remain since no one was redundant. Figure 4.1(C) shows the projected tree of φ_2. Here, too, the public push action executed by φ_1 in the team plan is removed. Furthermore, as φ_2 operates identically in both subtrees of the team plan, we remove the sensing action at the root of the team plan.

Team planning often builds a team solution like the one shown in Figure 4.1(A), which is not executable by the agents independently, and hence QDec-FP is bound to backtrack. Such team solutions schedule an agent φ_j to act differently based on the value of some proposition p observed by agent φ_i, even if φ_j cannot sense p. This is possible as QDec-FP treats all agents as part of one agent, having access to all the observations. For example, in Figure 4.1(A), even if φ_2 cannot sense the heavy box, the results of φ_1 sensing the heavy box are available to φ_2. This means when the agent φ_i senses p, then the *knowledge predicate* corresponding to p, i.e. Kp [1], holds immediately for all other agents, φ_j, too. Then, φ_j is viewed as knowing that a precondition p of its action(s) hold because of the sensing action of φ_i. In this example, φ_2 pushes the heavy box jointly with φ_1. Therefore, in a nutshell, team planning relaxes the need to maintain different information states for each agent. Instead, it manages just one belief state for all the agents.

However, there are many scenarios where agents may require to act under the conditions they cannot observe. In such scenarios, one agent should be able *communicate* certain information to another agent. Consider an example in which an agent keeps the room key underneath the doormat to share the information with another agent that the room is locked, which the second agent cannot sense. Similarly, the second example is when an agent moves the flowerpot inside the meeting room to share the knowledge with another agent that it is too sunny outside. In these examples, agents aim to share information by changing the state of the world. However, currently, it is practically not possible to support communication (either in an *implicit* form or in an *explicit* form) between agents since team planning considers that if "something" is known to one agent, it is known to all.

4.3 QDec-FPS – Approach Overview

We will describe a new algorithm, QDec-FPS, that uses enhanced reasoning about the knowledge of *individual* agents in the team problem. Reasoning about individual agents' knowledge during team plan execution has two important advantages: First, it will lead to the generation of more informed team plans that are easier to extend to true solutions, because the team planner adds an action only if the agent that executes it knows that its preconditions hold. Second, it allows us to model, within the team plan, the process of explicit and implicit communication, which we refer to here as *signaling*.

As we stated in the previous section, *signaling* refers to a case where agent φ_i can sense p but agent φ_j cannot. φ_i communicates this information to φ_j by setting the value of some variable q that agent φ_j can sense, to be correlated with the value of p. For example, φ_j cannot sense whether a door is open, but it can sense whether the light is on, while agent φ_i can sense both. If φ_i can also turn the light on and off, it can signal the door state to φ_j by turning on the light if and only if the door is open.

Technically, signaling consists of the following steps: (1) Agent φ_i senses p. (2) Agent φ_i sets the value of q to the value of p. (3) Agent φ_j senses q. To be sound, this behavior must be consistent between the two execution branches that follow the sensing of p: if p is *true*, we must ensure q is *true*. If p is *false*, we must ensure q is *false*. Note that, to support signaling, the planner must model the knowledge of each agent within the team plan. For otherwise, the planner has no reason to insert signals into the plan because signaling does not enhance the knowledge of the team. Moreover, some problems cannot be solved without signaling.

4.4 The QDec-FPS Planner

As we described in Chapter 3, to implement QDec-FP, one needs a single-agent off-line contingent planner. QDec-FP uses CPOR [43], which generates a policy tree by repeatedly calling the online contingent planner SDR [81]. SDR generates a single execution branch that corresponds to the true initial state. CPOR simply calls it multiple times with different "true" initial states.

While CPOR is the most scalable offline contingent planner currently, it has a major weakness: it does not backtrack. To overcome this, QDec-FP introduced a backtracking mechanism on top of CPOR. It modifies the planning problem with an additional, sound, constraint, that invalidates previous solutions. However, this limits the number of backtracks that are practically possible.

QDec-FPS has the same high-level structure as QDec-FP, but modifies the way the team problem is solved, and adds a mechanism for generating macros that enable signaling. The use of these macros is needed to overcome the limitations of the underlying CPOR single-agent solver.

4.4.1 Agent Specific Knowledge

QDec-FP uses the SDR translation from contingent planning to classical planning. This translation maintains, for each proposition p, two propositions: Kp and $K\neg p$, denoting knowing that p is true or false, respectively. SDR then transforms each precondition p of an action to Kp. The translation ensures that Kp holds in a belief state only if p holds in all possible worlds. This is done by reasoning about all the specific possible worlds explicitly.

The state description in the classical translation contains the current state of the world given every possible initial state, in the form of $Kp|s$, where p is a proposition, and s is a possible state. It also contains actions that allow deducing new knowledge facts, called *merge* and *refutation* actions [65]. Thus, the agent can also obtain Kp if for every currently possible state s, $Kp|s$ holds.

As QDec-FP solves the team problem as a single-agent planning problem, the planner treats all agents as part of a single centralized agent, and the knowledge of this agent reflects the combined knowledge of all agents. Thus, when one agent observes that p holds, other agents can use this knowledge, without observing the value of p themselves. This is inconsistent with the real plan tree, where each agent must independently ensure that p holds, before executing an action that requires p as precondition.

QDec-FPS changes the translation used by SDR to be agent aware. Instead of the Kp propositions that denote combined knowledge, it uses propositions of the form $K_{\varphi}p$, denoting that agent φ knows that p holds. The precondition p of an action executed by agent φ is replaced with $K_\varphi p$. The effect of a sensing action executed by φ is that only the sensing agent φ knows the value of p, i.e., either $K_\varphi p$ or $K_\varphi \neg p$. The same holds for the merge and refutation actions, which now provide agent-specific inference. However, the effects of non-sensing actions are still made known to all agents. We can understand this as a consequence of the fact that the agents know the policies of all other agents (or equivalently, they know the team plan).

This modification forces the underlying planner to insert sensing actions by different agents to ensure they have the knowledge required to perform their actions, whereas in QDec-FP, because all agents are treated as one, it was enough if some other agent performed this sensing action. If an agent has an action with a precondition p, the team plan will ensure that the agent first senses or learns the value of p. If the agent cannot sense or learn the value of p, such an action will not be part of a generated team plan.

This leads to the generation of team plans that better account for agent abilities. Like team plans in QDec-FP, these plans may not be extendable because they can require agents to act differently depending on information they do not have. Yet, they are much better informed, and are therefore less likely to lead to failure when agents extend the team plan. Hence, they often lead to fewer backtracks.

The new translation also allows us to solve problems that QDec-FP cannot solve. For example, imagine that Agent φ_1 can observe only p, and agent φ_2 can only observe q, but must execute an action with precondition p. QDec-FP's team plan will most likely let φ_1 sense p, and will make φ_2 execute the action afterwards. This team plan cannot be later extended by φ_2 to a valid local plan because it cannot sense p. QDec-FPS will avoid this team plan because it will know that φ_2 does not know p. Furthermore, as we discuss below, if φ_1 can set the value of q to be correlated with p, it will be able to exploit this implicit form of communication.

The new translation is more demanding and generates classical planning problems that are harder to solve. Because of this, it is able to detect unsolvable problems faster. However, in the special case of agents with identical capabilities, some problems are solved by QDec-FP while QDec-FPS times-out. In these problems, QDec-FP generates a simple team solution that lacks many sensing actions. However, since all agents have identical capabilities, it is easy to add these additional sensing actions into the projected single-agent planning problems. This simple team solution of QDec-FP is not a legal solution to the more complex team problem QDec-FPS generates. Solving this more complex problem requires numerous backtracks, making it unsolvable in the given time. In a sense, QDec-FP solves a more abstract, and hence a simpler problem. When most abstract solutions are extendable to full

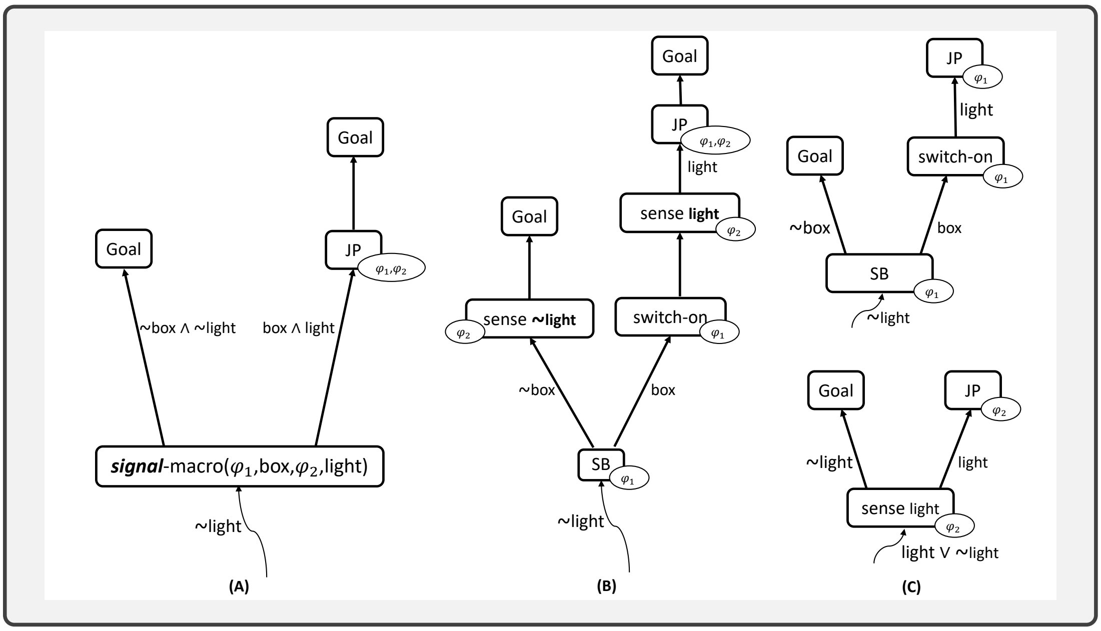

Fig. 4.2 *Signaling* in the QDec-FPS planner. (A) A team plan with the signaling macro action. (B) The team plan following the macro expansion. (C) Projected trees for φ_1 (top) and φ_2 (bottom).

solutions, as in the case of homogeneous agents, this is more efficient for large problems. When the problem's structure is more complicated, as in the case of heterogeneous agents, many solutions to the more abstract problem cannot be extended to a full solution.

4.4.2 Signaling

If agent φ_i can sense p but agent φ_j cannot, φ_i might be able to communicate p's value to φ_j. It could do this directly, if an explicit communication action exists, or indirectly, by setting the value of some variable q that agent φ_j can sense, to be correlated with p's value. In fact, explicit communication can be viewed as correlating the value of a channel variable with p. Signaling consists of the following steps (1) φ_i senses p. (2) φ_i sets the value of q to the value of p. (3) φ_j senses q. (4) φ_j reasons about the value of p.

Notice that (2) is not a restriction on a single branch of the plan, but a restriction on a sub-tree. φ_i must ensure that $p \leftrightarrow q$, which means that it needs to act differently in the branch where p is true and in the branch where p is false.

To operationalize this idea, we suggest to model the signaling process as a macro. In our context, macros are not simply a sequence of actions, but rather a part of a sub-tree.

To construct such macros, we must first discover the possible signaling options. We preprocess the domain seeking quadruples $(\varphi_i, p, \varphi_j, q)$ such that (1) φ_i can sense p but φ_j

cannot. (2) φ_j can sense q. (3) φ_i can modify the value of q. For simplicity, we consider only propositions q that can be affected by a single action that does not change the value of any other public proposition. This can be extended to more complex sub-plans for modifying the value of q. For each such quadruple, we add the macro-action $signal(\varphi_i, p, \varphi_j, q)$. Notice that this process is problem-instance *independent* and can be done in linear time.

The macro $signal(\varphi_i, p, \varphi_j, q)$ is treated by the planner as a sensing action that has two possible outcomes: in one of them $K_{\varphi_i} p \wedge K_{\varphi_j} q$ holds, and in the other $K_{\varphi_i} \neg p \wedge K_{\varphi_j} \neg q$ holds. Unlike the pure sensing actions we use, this action changes the state of the world, as well, ensuring that this correspondence between the values of p and q will hold.

Given a team plan with a signaling macro, we first expand the macro as indicated above: First, φ_i senses the value of p. For each of the two resulting branches, it must ensure that q's value is appropriately correlated, by applying actions that affect q's value appropriately. Then, we add in each branch a sensing action a_q where φ_j senses the value q. While regular sensing actions have two possible children, a_q has only a single child in the team plan because, at the team level, its outcome is known. When projected to φ_j's local plan, however, a_q appears like a regular sensing action. This macro expansion is described in Figure 4.2.

In the next step, the projection of the team plan containing the macro expansion is solved by each agent. This requires, in particular, that agent φ_i will change the value of q as needed in each branch, and that both agents perform their sensing actions — φ_i over p and φ_j over q.

Figure 4.2 shows an example of this process. Two agents, φ_1 and φ_2 must jointly push a heavy box. φ_1 can sense the box, but φ_2 can only sense whether a light is on. φ_1 can signal to φ_2 about the box by turning on the light, which is originally off. Figure 4.2A shows a team plan with a macro. Figure 4.2B shows the team plan after macro expansion: φ_1 senses the box, then turns on the light, if needed, and then φ_2 senses the light. Finally, both jointly push the box. Figure 4.2C shows the projected single agent plan trees for φ_1 and φ_2.

In practice, adding this macro on top of an online contingent solver that focuses on a single branch at a time, such as SDR, is not straightforward. To address this, we do the following: We add an action by φ_i that can be viewed as a commitment to ensure that $p \leftrightarrow q$ holds. This action is constrained to be followed immediately by the action of sensing p by φ_i. At this point, in the team plan, if agent φ_j needs to know the value of p, it can use the fact that $p \leftrightarrow q$ to deduce it from the value q. To ensure that it learns the value of p, we force the action of sensing q in both branches. As above, the team plan is post-processed to ensure that φ_i does indeed ensure the validity of $p \leftrightarrow q$ following the sensing action.

4.4.3 QDec-FPS Properties

We discuss the soundness and completeness of the QDec-FPS algorithm. But, when discussing soundness and completeness, we should first distinguish between the abstract approach used by QDec-FP/S, which reduces multi-agent contingent planning to single-agent contingent planning, and the practical implementation of the planner which uses a specific single-agent contingent planner, CPOR [43].

Theorem 3 *The QDec-FPS algorithm is sound if the underlying single-agent contingent planner is sound.*

Proof First, consider the abstract formulation and assume a sound and complete underlying single-agent contingent planner. The soundness of QDec-FPS without signaling follows from that of QDec-FP: Every team solution of QDec-FPS is also a possible team-solution for QDec-FP and all the other steps are identical. This remains true in the implementation due to the soundness of CPOR.

With signaling, the abstract version of the macro is a sensing action with effects. It remains sound if the action used by the signaling agent to ensure that $p \leftrightarrow q$ holds does not affect any other proposition. The implementation using CPOR is more complex, but it ensures that following the commitment to the signaling macro, the team solution implements a correct version of the macro, and hence it is sound, too. □

Next we explain why the QDec-FPS algorithm is incomplete even if the underlying single-agent contingent planner is complete.

For QDec-FPS, one cannot separate between an abstract and implemented case because the core contribution of QDec-FPS is at the level of the implementation of the classical encoding used by CPOR and SDR. At this level, QDec-FPS is incomplete for two reasons: First, CPOR is an incomplete solver: it lacks an internal backtracking mechanism. [1]

Second, the planner's ability to reason about the knowledge of separate agents, the way it is implemented, at present, is incomplete. That is, it can underestimate the knowledge of an agent. Thus, it might not be able to deduce that the preconditions of some actions are known to the acting agent, even though they are. An example of such a problem is provided next.

In Example 3, we illustrate a case where the team planning of QDec-FPS is unable to deduce that a precondition of an action is known to the acting agent, even though this precondition is known.

[1] For that reason, the implementation of QDec-FP is also incomplete, although the abstract formulation is complete. Note that team plans containing signaling are legal team plans for QDec-FP and thus, in theory, do not impact its completeness. In practice, however, such plans are extremely unlikely to be generated.

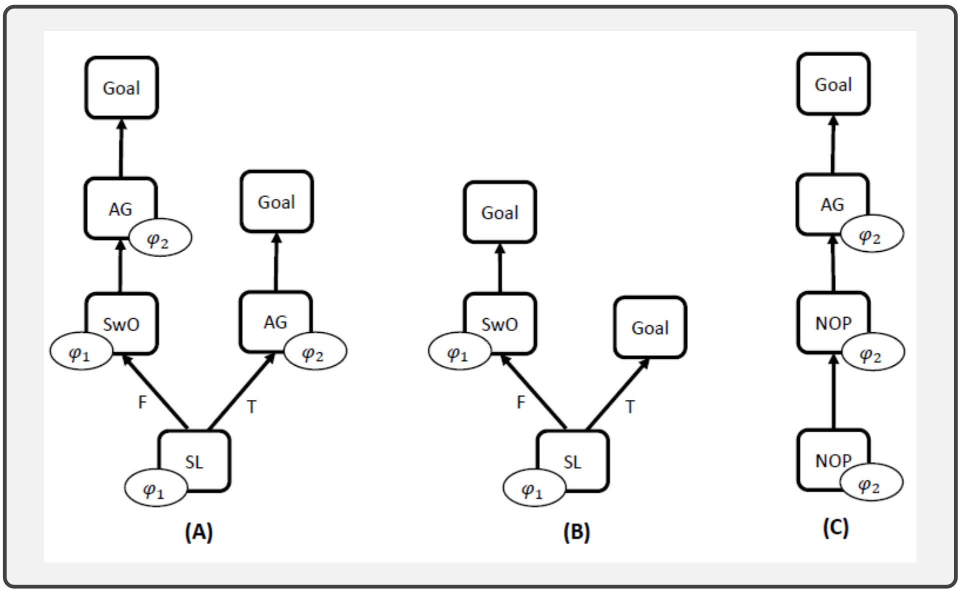

Fig. 4.3 A *legal* team plan for the team comprising φ_1 and φ_2 is shown in Sub-Figure (A). The individual policies: for φ_1, it is shown in Sub-Figure (B), and for φ_2, it is shown in Sub-Figure (C). Here, *SwO* refers to the action *switchOn*, *SL* is *senseLight*, *NOP* shows the *noop* action, and *AG* is *achieveGoal*.

Example 3 *Consider a domain that models two agents φ_1 and φ_2, and* light *such that only φ_1 can* sense light*, and only if the* light *is off, it can* switchOn *it, i.e., it has an action switchOn whose only precondition is $\neg$light and effect is light. While φ_2 needs only light as a precondition to execute its only action* achieveGoal*, whose effect is* goal*. The initial state uncertainty is $(light \vee \neg light)$ and initially* ¬goal *holds, while the goal condition is* goal.

For Example 3, we can generate a legal team plan and the individual policies based on that team plan (following the notations used in Chapter 2), which are shown in Figure 4.3, for the agents φ_1 and φ_2.

First, let us discuss how the QDec-FP solver would handle this example problem. The *relaxation* made about the agents' knowledge of the world during team planning helps deduce that the precondition of *achieveGoal* is known to φ_2, which eventually achieves the goal, as shown in Figure 4.3(A). At the second stage of the QDec-FP approach: the new contingent problem based on the projection obtained for φ_1, schedules φ_1 to apply *switchOn* when $\neg light$ holds, while in the other branch, but when *light* is true, *NOOP* (for a reference, see Figure 4.3(B)). Similarly, the new contingent problem obtained for φ_2 forces it to apply the action *achieveGoal* only, as the sensing action at the root of the team plan is redundant (Algorithm 1). For aligning the solutions, a couple of *NOOPs* are added (for a reference, see Figure 4.3(C)).

A legal team plan is shown in Figure 4.3(A). In this team plan, φ_1 senses *light* and when it holds, φ_2 applies *achieveGoal* to achieve the goal. One can see that, in the case of QDec-FPS, the agent φ_2 cannot deduce the required knowledge to apply the *achieveGoal* action when there is *light*, even though φ_2 has this knowledge. The reason is that φ_2 can deduce that *light* holds in this particular branch, but it is possible only when this agent considers the entire plan tree (*i.e.*, both these branches). However, this deduction cannot be carried out by φ_2 considering only this particular branch alone, and which is what is done by the SDR solver. Hence, SDR underestimates the knowledge of φ_2 in this branch, and as a result, the team planner would fail to find a legal team solution, which implies that QDec-FPS returns a failure.

4.5 Empirical Evaluation

We now examine the applicability and scalability of QDec-FPS by comparing it with QDec-FP on *three* multi-agent planning domains. Some problems in each domain were modified to require *signaling*. Both QDec-FP and QDec-FPS are implemented in C#, and were run on a Windows 10, 64 bit machine with *i*7 processor, 2.8GHz CPU, and 16Gb RAM. IMAP was not considered, as QDec-FP was shown to scale better than IMAP [5].

4.5.1 MAP Domains

We experiment with the following three domains.

Box-Pushing (BP)

We described this domain in Chapter 3, Section 3.6.1.

Table-Mover (TM)

This domain consists of several tables, rooms, and agents that can move between connected rooms (for a detailed explanation of TM under a more classical setting, see [83], Section 6.1.2, or Chapter 5, Section 5.7.1). The locations of the tables are uncertain, initially. The agents must move each table to its dedicated goal locations. Like in the Box-Pushing domain, agents in the Table-Mover domain are *non-homogeneous*, too. Different agents can sense the presence of different tables. We model TM such that each table has some fragile items on top of it, and these objects must remain intact when the tables appear at their destinations. To achieve this, agent must perform collaborative actions to manipulate the table, for example, collaborative actions like *2move-table*, *2lift-table*, *2drop-table*. The actions *2lift-table* and *2drop-table*, indicate that two agents, respectively, lift and drop a table simultaneously, keeping the objects on the table intact in the process.

Rovers

We described this domain in Chapter 3, Section 3.6.1.

4.5.2 Experimental Results

Table 4.1 compares QDec-FPS and QDec-FP based on policy quality (*max-width*, *max-height*), runtime (*time*), and the number of *backtracks* required. *max-width* and *max-height* refer to maximum number of branches and the maximum height of all individual solution trees obtained for the agents. The number of branches is also indicative of the number of sensing actions performed, as branching occurs following an observation. The planner backtracks when at least one of the single-agent problems, obtained by decomposing τ_{team}, is unsolved by CPOR. Within each domain in the table, dashed lines separate three problem classes: homogeneous agents, non-homogeneous agents, and non-homogeneous agents that require signaling. To handle signaling, QDec-FPS adds macros, which may have a significant overhead. Therefore, these macros were only added in problems that require signaling. The decision whether to add macros was done *manually*. In the future, we will automatically detect whether signaling is needed.

In BP, QDec-FPS scales much better than QDec-FP in problems with three to five agents. Increasing the number of objects had minor impact on QDec-FPS running time, as opposed to QDec-FP. For many problems, QDec-FPS needs to backtrack fewer times than QDec-FP, and as a result, it finds solutions faster. On the other hand, increasing the number of agents has an adverse effect on QDec-FPS. In fact, instance B3 in the BP domain with nine *identical* agents, was quite rapidly solved by QDec-FP, while QDec-FPS times out. This is because the new translation makes the team problem much harder to solve. Thus, QDec-FP finds a team plan quickly, and when agents are identical, it is usually easy to fix the team problem by adding any needed sensing action and no or few backtracks are needed. On the other hand, problems B11 and B12 were solved by the QDec-FPS planner but were unsolved by QDec-FP. This is most likely due to the high number of required backtracks. We can conclude that for simpler problems with identical agents, QDec-FP scales better with the number of agents, but for more complex problems, QDec-FPS is required.

Unlike BP, in TM each public action is a collaborative action. Similar to the BP domain, when backtracking is not required to solve an instance, for example, the simpler instances T1 to T5, the QDec-FP solver takes less time to solve it than that of QDec-FPS. When problems are more complex and backtracking is required, e.g., for the instances T6 and T11, QDec-FPS is faster.

In Rovers, we see a similar trend as BP and TM. For the simple instances like R1 to R7, the performance of the planners is mixed. When backtracking maybe required since the rovers are non-homogeneous (instances R8 to R13), QDec-FPS outperforms QDec-FP on all instances. Instances with more objects are solved relatively easily by QDec-FPS.

Problem T9 is interesting because it is unsolvable. This requires the planner to rule out all possible solutions. Because there are fewer solutions to the team problem QDec-FPS generates, it was able to conclude that no solution exists much faster than QDec-FP using six backtracks, compared to 31 for QDec-FP.

The table clearly shows that QDec-FPS generates smaller trees across all domains. A closer inspection of the policies also shows that the number of *noops* in its solution is smaller.

To test signaling, we added new instances to all domains that cannot be solved without signaling (B13-B19, T12-T16, R14-R22). These instances cannot be solved by QDec-FP (shown as *na*). In the BP instances, moving from 2-3 agents to 4 increased runtime by an order of magnitude. This is likely due to the number of optional pairwise signaling added with each agent. However, adding objects did not much impact runtime.

In the TM domain, we see a more mixed picture. Both adding agents and objects can increase run-time, although not always (e.g. T14 vs. T12 and T13). Problem instance T14 is particularly interesting because it can be solved without signaling (hence QDec-FP solves it with 9 backtracks), but it is solved even faster with signaling. For this instance there are three agents, φ_1, φ_2, and φ_3 such that φ_1 and φ_3 together are capable of solving this problem without signaling. As we purposely placed φ_3 farther from the table's location, QDec-FPS generated a plan where φ_1 signaled to φ_2 the table's location and they achieved the goal together, without φ_3. This solution was generated much quicker and with no backtracks.

Signaling in Rovers is similar to BP. Adding more agents makes the problem harder, especially when we move to 4 agents, whereas the effect of objects is less clear.

4.6 Summary

The new planner, QDec-FPS, uses a factored approach, introduced in QDec-FP, to solve MAP problems by solving a set of single-agent planning problems. By reasoning about the knowledge of individual agents, QDec-FPS can generate more informed team plans, which provide better skeletons for the final solution plan, and lead to fewer backtracks. QDec-FPS shows better applicability and scalability than its counterpart. Moreover, the ability to reason about individual agents' knowledge allows it to support model communication and planning using that. In this work, we supported, *Signaling* — a form of implicit communication, which takes place through the state of the world. Signaling allows agents to share information and thus enables solving planning problems that were not practically solvable by QDec-FP.

Although QDec-FPS scales overall better than QDec-FP on deterministic QDec-POMDP problems, extra work is required to make QDec-FPS applicable to complex domains like non-deterministic MAP domains.

Domain	Ins (#agt)	Objects	Max-width		Max-height		Time (sec)		BT	
			fp	*fps*	*fp*	*fps*	*fp*	*fps*	*fp*	*fps*
BP	B1 (3)	16	8	5	23	18	3.59	**2.91**	0	0
	B2 (4)	16	12	10	19	19	**5.3**	6.1	0	0
	B3 (9)	36	64	*	24	*	**25.3**	*	0	*
	B4 (3)	11	4	4	14	11	16.39	**1.17**	9	0
	B5 (3)	12	6	8	16	15	13.65	**2.9**	4	0
	B6 (3)	12	-	6	-	12	-	**13.58**	47^+	4
	B7 (3)	12	8	8	18	19	158.89	**3.87**	41	0
	B8 (3)	12	8	8	17	21	111.6	**4.05**	26	0
	B9 (3)	13	16	14	21	19	121.42	**5.6**	19	0
	B10 (3)	16	16	15	26	29	155.83	**9.69**	33	0
	B11 (5)	20	*	24	*	32	*	**75.21**	*	1
	B12 (5)	20	*	24	*	37	*	**365.9**	*	6
	B13 (2)	10	*na*	2	*na*	6	*na*	**1.06**	*na*	0
	B14 (2)	12	*na*	2	*na*	2	*na*	**1.20**	*na*	1
	B15 (3)	12	*na*	4	*na*	12	*na*	**2.49**	*na*	0
	B16 (3)	13	*na*	4	*na*	7	*na*	**5.5**	*na*	1
	B17 (3)	13	*na*	4	*na*	6	*na*	**8.9**	*na*	2
	B18 (3)	14	*na*	7	*na*	14	*na*	**4.04**	*na*	0
	B19 (4)	15	*na*	8	*na*	14	*na*	**17.6**	*na*	2
TM	T1 (3)	10	8	7	20	16	**2.68**	2.78	0	0
	T2 (4)	12	16	13	21	17	**7.84**	8.26	0	0
	T3 (4)	15	12	16	34	26	**8.74**	12.39	0	0
	T4 (5)	14	19	22	22	24	**20.5**	43.58	0	0
	T5 (5)	16	12	14	32	25	**9.7**	11.2	0	0
	T6 (3)	8	2	2	8	8	11.01	**0.61**	7	0
	T7 (3)	10	8	7	20	16	**34.5**	41.8	8	8
	T8 (3)	10	8	8	24	22	37.87	**22.45**	9	4
	T9 (3)	10	-	-	-	-	140.32	**29.73**	31	6
	T10 (4)	12	16	16	23	22	**6.68**	152.18	0	14
	T11 (5)	16	16	16	30	35	274.5	**56.15**	27	3
	T12 (2)	10	*na*	2	*na*	9	*na*	**1.7**	*na*	0
	T13 (2)	12	*na*	4	*na*	17	*na*	**6.81**	*na*	0
	T14 (3)	12	2	2	16	11	17.59	**2.32**	9	0
	T15 (3)	13	*na*	4	*na*	14	*na*	**7.80**	*na*	0
	T16 (3)	14	*na*	4	*na*	19	*na*	**11.68**	*na*	1
Rovers	R1 (1)	12	2	2	8	8	3.8	3.8	0	0
	R2 (2)	14	2	2	9	8	5.4	**5.3**	0	0
	R3 (2)	13	4	4	15	15	9.3	**8.9**	0	0
	R4 (2)	17	12	12	34	23	**35.8**	38.6	0	0
	R5 (3)	18	12	12	28	32	**51.5**	54.1	0	0
	R6 (3)	21	30	30	36	29	136.6	**111.9**	0	0
	R7 (3)	23	98	81	50	45	567.9	**233.2**	0	0
	R8 (3)	13	2	2	7	6	57.9	**5.9**	3	0
	R9 (3)	14	4	4	12	12	113.8	**10.2**	4	0
	R10 (3)	14	4	4	11	11	314.5	**89.3**	11	3
	R11 (3)	19	-	12	-	12	325.1	**41.1**	6+	0
	R12 (4)	15	4	4	12	11	47.2	**16.0**	1	0
	R13 (4)	20	12	12	25	19	241.8	**49.1**	3	0
	R14 (2)	16	*na*	2	*na*	9	*na*	**3.6**	*na*	0
	R15 (2)	18	*na*	2	*na*	10	*na*	**5.89**	*na*	0
	R16 (2)	16	*na*	4	*na*	13	*na*	**34.17**	*na*	0
	R17 (2)	20	*na*	4	*na*	14	*na*	**17.68**	*na*	0
	R18 (3)	17	*na*	2	*na*	7	*na*	**10.17**	*na*	0
	R19 (3)	18	*na*	4	*na*	13	*na*	**12.93**	*na*	0
	R19 (3)	14	*na*	2	*na*	7	*na*	**6.62**	*na*	0
	R20 (3)	19	*na*	8	*na*	16	*na*	**36.74**	*na*	0
	R21 (4)	20	*na*	6	*na*	11	*na*	**67.79**	*na*	0
	R22 (4)	22	*na*	6	*na*	12	*na*	**68.21**	*na*	0

Table 4.1 Comparison of QDec-FP and QDec-FPS planners. *Ins (#agt)*: instance number and number of acting agents. *Object*: the number of objects in each problem. *BT*: number of planner backtracks. *: time out. '-': planner could not solve or breaks down. *na*: problem not applicable to a solver. *fp*: QDec-FP. *fps*: QDec-FPS. The best approach, based on time only, is shown in bold.

Chapter 5

Representing and Planning with Interacting Actions and Privacy

Publication(s)

1. Shashank Shekhar and Ronen I. Brafman, Representing and planning with interacting actions and privacy, Artif. Intell. 278, (10.1016/j.artint.2019.103200) (2020).

2. Shashank Shekhar and Ronen I. Brafman, Representing and planning with interacting actions and privacy, in Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018, pages 232-240.

5.1 Motivation and Overview

In Chapters 3 and 4, we evaluated our approaches for efficiently solving QDec-POMDPs on several MAP benchmark domains. Consider the nature and structure of the MA problems that the QDec-POMDP approaches managed to solve. One main property of these problems is that agents' policies are only loosely-coupled. Moreover, we modeled them such that, at one time instant, only one *public* action can be executed that is by either a single agent or a set of agents. The actions applied by multiple agents together, i.e., the collaborative actions [5], which are always public, and a single-agent action is public if its execution affects or gets affected by another agent's actions, or if it achieves some goal (for more details on this, see [14]). In post-processing phase, as in [23], one could parallelize a generated plan tree by the QDec solvers as much as possible to make the plan tree more compact.

But in order to be able to generate a more compact solution, we must understand what happens when multiple single-agent actions are executed concurrently. Note that being able to compress a plan and schedule more actions together is not necessarily needed just for efficiency purpose. In problems where there are deadlines, or in problem where there may be limited resources (e.g., a boat can only be used once), it may be necessary to combine many actions together to meet the deadlines, or to optimally exploit the resource. In general, we may need to try to schedule as many actions as possible concurrently. But how do we model the effect of diverse combinations of single-agent actions?

Of course, we could explicitly specify the effect of every combination of actions in a brute-force manner. But this is unrealistic with a large number of agents, as the number of single-agent combinations, i.e., joint-actions, is exponential in the number of agents. For this reason, we ask: can we somehow deduce the effect of concurrent execution of actions from the description of single-agent actions and some useful collaborative actions?

To see this, consider a planning domain that models a single-agent action, *lift(table)*, and a two agents collaborative *joint-lift(table)*. In that case, the planner can schedule two single-agent *lift(table)* actions by two agents, which should be *illegal* as there exists a *joint-lift(table)* action. Also, suppose that there exists a collaborative action, *joint-lift(table)*, comprising *three* different agents. Now, the planner must be forbidden to apply three single-agent *lift(table)* actions together, or a two-agent *joint-lift(table)* and a single-agent *lift(table)* actions together.

The primary contributions of this chapter is an intuitive formalism for specifying joint-actions in a compositional way and the definition and empirical evaluation of a compilation-based approach to planning by teams of agents with interacting actions, as well as privacy preserving versions of this model, for which we introduce a number of new domains. Besides, this chapter highlights and discusses subtle issues that arise when attempting to model and plan with interacting actions

5.2 Introduction

What happens when multiple agents perform actions concurrently? In principle, every combination of actions performed concurrently by a group of agents – a *joint action* – may define a different state-transition function. But as the number of joint-actions is exponential in the number of agents, specifying an explicit model for each combination of single-agent actions is impractical except for very simple cases.

To be succinct, a representation for joint actions must be compositional. That is, there must be some way of deducing the effect of the concurrent execution of actions $a_1,\ldots,a_n$ from the effects of smaller combinations. One option is to use a logical language and describe the effects of such combinations via formulas in this language. But for this, classical logic does not suffice. If we want each set of formulas to yield a unique model (that is, to specify a unique transition function), we must use some sort of non-monotonic logic, such as [4, 48, 67]. While this might yield a satisfying representation scheme, it makes planning exceedingly difficult – indeed, most planners have difficulty even handling classical logic deductions.

Thus, the primary challenge for planning for multi-agent systems with interacting actions is finding a model for joint actions involving large sets of agents and large sets of single-agent actions, that is both succinct in "natural" settings and supports efficient planning. We believe that a key requirement for the latter is monotonic behavior – as the planner gradually inserts additional components to a joint-action, the effects of added components should not undo the effects of components inserted previously. To achieve this, we will construct joint-actions from a richer set of components, and will also restrict the components that can be added to it at each point in time. This will be made clearer later.

Due to the difficulty of representing and planning with interacting actions, most work on multi-agent planning algorithms ignores this issue, and considers sequential actions or concurrent non-interacting actions. In the former case, joint-actions are not an issue, and sequential plans containing actions by different agents are generated. In the latter case, only actions that impact different variables, or have the same effect on shared variables are considered. In that case, the effect of a joint-action is the union of effects of its single-agent components. We will follow a similar approach, except that in our case, a joint-action will be composed from a set of components that is richer than single-agent actions, allowing us to model the full spectrum of possibilities.

While much can be achieved without considering joint interacting actions, there are many settings where agents must coordinate their actions carefully to obtain desirable effects: a single-agent may be unable to lift or push heavy items, whereas this is possible for multiple agents acting together; if a table is not lifted from both sides concurrently, objects on it will fall; in robot soccer, more advanced teams perform coordinated maneuvers, such as one agent passing the ball to a free region while the intended receiver moves to this area at the same time; and in more complex manipulation tasks, coordinated activities of two or more arms are needed. In these examples, the effect of each move on its own is quite different from the effect of the combined actions.

To define the effect of joint-actions, we introduce *collaborative actions*. A collaborative action is a minimal combination of single-agent actions that cannot be defined as the union of its components. A joint action is defined using a *well-formed* set of non-interacting single-agent and collaborative actions. Roughly speaking, a joint-action is well formed if its components (single-agent and collaborative actions), or parts of its components, cannot be combined to yield more complex components. For example, consider box-pushing agents. A single-agent *push* action is effective when the box is light. A two-agent collaborative action *2push* – composed of two concurrent single-agent *push* actions – is effective when the box is heavy, as well. Given this, a joint action consisting of two single-agent *push* actions is not well-formed because these actions can be combined to form the collaborative *2push* action.

The difficulty of planning with joint actions depends on what interactions are allowed, and whether a distributed and privacy preserving algorithm is required. In the simplest case, only non-interacting concurrent actions are allowed in order to reduce make-span. A slightly more interesting case is when concurrent actions can destroy each other's preconditions. For example, suppose that a building becomes locked once an agent is detected entering it. Here, sequential execution allows a single-agent to enter the building, but parallel execution allows more agents to enter the building. In both cases, the effect of concurrent execution is the union of effects of the single-agent actions. Thus, there is no representational issue. But while sequential planning with post-processing works in the first case, a sequential planner cannot insert two *enter-building* actions. More complicated is the case where the effects of actions performed together differ from the union of their effects. Finally, on top of each of these cases, one can introduce the goal of preserving privacy. We will describe compilation methods that support all cases, as well as, the notion of object cardinality constraints, introduced by Crosby, Jonsson, and Rovatsos (2014).

The source code and the domains we used are all available. [1]

[1] GitHub: https://github.com/shekharsai/bgu-upf-map/. This code builds on the software of Furelos-Blanco, Frances, and Jonsson (2019) which implements and extends the techniques of Crosby, Jon-

5.3 Related Work

Most work in classical planning allows for concurrent non-interacting actions in order to reduce the plan make-span and, in some cases, the depth of the search tree (see e.g., [9]). Actions can occur concurrently if they do not interact with each other. However, there is no real notion of multiple agents, and the accepted semantics of this "concurrency" is that actions can be executed in any order with the same effect. A sufficient condition for this is that the union of effects and preconditions of concurrent actions is consistent. An alternative, called the $\exists$ semantics [25, 75] was studied in the context of planning as satisfiability. It allows for interacting actions, provided they can be ordered in some sequentially legal way. But this semantics does not address true parallelism. Rather, it is a means of generating plans with fewer steps, for search efficiency reasons. It can be ambiguous, when multiple orderings are possible, and moreover, some natural examples described later do not have even a single legal sequential ordering.

Recently, Crosby, Jonsson, and Rovatsos (2014) (henceforth, CJR) introduced an approach for centralized planning for multi-agent systems. In their formalism a joint-action is executed in each time step. The joint-action contains one single-agent action per agent, where that action can be a *no-op*. The preconditions and effects of a joint-action are the union of the preconditions (and resp. the effects) of its single-agent actions. In addition, they introduce a limited form of interaction among actions through object cardinality constraints. Every action is associated with a set of objects, and every set of objects may have constraints limiting the number of agents that can manipulate these objects concurrently. For example, a ship can sail if at least two agents sail it concurrently, or a bridge can be crossed by at most three agents concurrently. A joint-action is applicable in a state s if (1) its preconditions hold in s; (2) its effects are consistent; and (3) it satisfies the cardinality constraints on objects that appear in it. This formalism captures a limited form of interaction via cardinality constraints in a natural manner, but still assumes that the effect of a set of concurrent actions is the combined effect of the single-agent actions in this set.

Boutilier and Brafman (1997) (henceforth, BB), were the first to extend STRIPS-like languages to address interacting actions and to propose an extension of a standard planning algorithm to handle such domains. This technique was later formalized in an extension of PDDL3.1 to multi-agent planning [45]. Very recently Furelos-Blanco and Jonsson (2019) [34] proposed an approach to collaborative planning based on representation given by BB.

BB's extension to STRIPS is conceptually simple: in addition to a list of preconditions, an action a has a concurrency condition that specifies which actions must or must not be

sson, and Rovatsos (2014) for compiling multi-agent planning with limited interactions, available at https://doi.org/10.5281/zenodo.2593129.

executed concurrently with *a* for *a*'s effects to hold. Effects can also be conditional on which actions are executed concurrently. For example, consider an action for lifting the side of the table. If performed by two agents on both sides, objects on the table will remain. But if performed by a single agent, the objects will fall. Thus, the action of lifting the left side of a table will have, beyond its regular preconditions and effects, a conditional effect with concurrent effect condition that states that when the action of lifting the right side is not performed concurrently, objects on the table will no longer remain on the table. Similarly, a table can be moved only if both agents holding its side move in the same direction. Thus, the action of moving the table north by one agent will have a concurrency condition that requires a move-north action by another agent. Naturally, that action will have a precondition that that agent is actually holding the table. Another example is box pushing – if the box is heavy and one agent pushes it, it remains in place. If two agents push it then it will move. Thus, the box movement is a conditional effect with a concurrency condition requiring another push action. Such an action can be modeled using the BB's extension of STRIPSas shown below.

```
PushHeavyBox(agent, box, location)
precondition: at(agent, location) & at(box, location) & heavy(box)
concurrency: PushHeavyBox(agent', box, location) & agent'!= agent
effect: NOT at(box,location)
```

To deduce the effect of a joint action, one must take the union of the effects of the individual actions (assuming they are consistent), where the effect of each individual action takes into account the other actions performed.

BB's method is clean and clear semantically, but it has some potential shortcomings: 1. It introduces the additional, non-standard, concurrency condition. 2. The list of conditional effects when interactions are more involved – especially if the effects are non-monotonic in the number of agents – can be quite complex, and its consistency must be ensured (as when complicated conditional effects are used). 3. Action schema generally require existential quantifiers in their specification. If the effect of action a changes when a' is concurrently executed, this usually holds for multiple instantiations of parameters of a'. These must be existentially quantified in the concurrency condition. For example, all actions have an agent parameters whose identity usually does not impact the interaction. 4. Their planning algorithm is based on partial order planning, a method not competitive with the state of the art. 5. An action's effects change once additional concurrent actions are introduced into the plan. This makes its behavior non-monotonic.

Brafman and Zoran (2014) consider an alternative formulation in which actions involving multiple agents (which we call *collaborative actions* here) are specified [19]. That work was preliminary, did not carefully consider the issue of subsumed actions (discussed later in this chapter) nor did it support concurrent actions that affect each other's preconditions. To support this new input language, the authors modified the MAFS algorithm [58] by introducing new messages types. In this chapter we provide a more careful and general definition and treatment of joint actions. Our compilation approach makes it easy to use off-the-shelf, state-of-the-art privacy-preserving planners, such as [53], which is appealing from the engineering point of view. In addition, we support object cardinality constraints and interacting actions that modify each others' preconditions, and provide a more complete experimental evaluation.

Earlier work in knowledge representation considered the issue of concurrent actions, too, e.g., [4, 48, 67]. These work focus on the representational issue only, and use non-monotonic formalisms, often within a rich first-order language such as the situation calculus [72]. As noted earlier, such formalism are hard to integrate efficiently within modern planning algorithms. Of these formalisms, it is worth noting the action language $\mathscr{A}_c$ [4] which we find the simplest and most intuitive. It is also closest semantically to our approach and uses propositional logic. In this language, the basic statements are of the form: "p is an effect of A if c". That is, if the actions in set A are executed concurrently in a state satisfying c then p will hold. This implies that p is also an effect of every $B \supseteq A$ given c, as long as there is no other set of actions D such that $B \supseteq D \supseteq A$ and $\neg p$ is an effect of D given c. This formalism, too, is non-monotonic – as when one adds actions to a set, effects that held for the subset may no longer be true. Our approach can be viewed as a monotonic variant of this semantics that forces the planner to be more explicit about the desired effects and restricts the extension of sets that might invalidate them.

5.4 Modeling Joint Actions

In this section, we first give an informal overview, which is followed by the formal definitions.

5.4.1 An Informal Description

We make the standard distinction between the language used to specify joint actions and the semantic notion of a joint-action. To illustrate the key ideas, we will use a running example of a domain with three agents on a one dimensional grid with boxes, where each agent can either move, push a box out of the grid, or do nothing, and boxes can be light or heavy. When an agent pushes a heavy box, it does not move out of the grid, but when two agents push it, it does move.

The basic assumption in most multi-agent planning models is that, at each clock tick, each agent makes some choice, and the world changes in accordance to the set of choices made by *all* acting agents. This leads to the two key semantic concepts: *single-agent signals* and *joint actions*.

- The set of *single-agent signals* describes the basic control signals each agent can select from at each time point. We will use $\mathscr{A}_i$ to denote the set of control signals available to agent φ_i, which will always contain the distinguished *no-op* signal. In our example domain, the single-agent signals are *move-left*, *move-right, push, no-op*.
- The set of *joints-actions*, denoted by A, corresponds to the possible combinations of choices made by the agents, i.e., to the Cartesian product $\mathscr{A}_1 \times \cdots \times \mathscr{A}_n$, where n is the number of agents. Each element of A is a vector of size n containing one signal for every agent. In our example, the set of joint actions correspond to three-tuples, such as *(move-left,push,push)*. In this latter tuple, agent φ_1 provides the signal *move-left*, φ_2 provides the signal *push*, and φ_3 provides the signal *push*.
- Each joint-action $a \in A$ is an action in the classical-planning sense: it is a (typically partial) mapping from the state before the action is applied, to the state following its application. We refer to this mapping as the *transition function for a.* As in classical planning, we focus on transition functions that are specified compactly using precondition and effect lists. For example, the joint action that corresponds to the tuple of signals *(move-left,push,push)* has a transition function in which the position of φ_1 changes one cell to the left, as long as it is not on the left side. If φ_2 and φ_3 are in the same cell, and that cell contains a box, it will move out of the grid. If they are located in different cells, and any of these cells contains a light box, the box will also move out of the grid. Otherwise, no box changes its position.

In multi-agent planning, the world changes following the application of a joint-action, exactly as in classical single-agent planning the world changes following the application of an action. But while in single-agent planning the actions are given as a primitive set, in multi-agent planning the joint-actions correspond to vectors of primitive choices available to each agent, which we referred to above as its possible *signals*.

The signals available to each agent are more commonly referred to as the *single-agent actions*, and we adapt this terminology. Technically, they are not really actions – they do not define a transition, but we can treat them as such if we map every single-agent action a_i of agent φ_i to the joint-action in which φ_i's signal is a_i, and the signal of all other agents is *no-op*.

The size of A is exponential in the number of agents. Explicitly enumerating the set of transition functions for each element of A is not realistic. Instead, we seek a language that can compactly and implicitly define all elements of A. One simple extreme case is when a joint action can be described as a simple combination of its single-agent actions, i.e., by taking the union of their preconditions and effects. Then, we need to only specify $\sum_{i=1}^{n} |\mathscr{A}_i|$ actions. The other extreme case is when the interactions are so complex and irregular, and we have no choice but to explicitly describe the transition function for each element of A.

In this chapter of the thesis we formally describe a language (*next subsection*) that gracefully increases in complexity depending on the level of interaction between single-agent actions. It specifies, using preconditions and effects, two types of actions:

- *Single-agent* actions. Each single-agent action is described under the assumption that no additional action is executed concurrently. The description of the single-agent assumes no other single-agent action is executed concurrently. For example, *push(a1,b)* in which agent $a1$ pushes a box b will include the precondition and effects of executing this action assuming no other action is executed concurrently.

- *Collaborative* actions. They describe the effect of a *set* of (more than one) single-agent actions executed concurrently. They, too, assume that no additional action is executed concurrently.

- Roughly speaking, the language should contain a description of a collaborative action only if the effect of this collaborative action is *different* from the union of the effects of the single-agent actions it contains. For example, the effect of two *push* actions is *not* the union of the effects of these actions because a single push action does not move a heavy box, while two concurrent push actions do. This requires us to add a collaborative action *2push* for every pair of agents that consists of a pair of single agent *push* actions and captures this interaction. When b is heavy, the effect of *2push*(a_1,a_2,b) is different from the union of the effects of the *push*(a_1,b) and *push*(a_2,b) actions it consists of.

To relate between this language and the semantic concept of joint-actions, we must define the notion of *multi-actions*. Multi-actions are not part of the language – the specification of a domain does not contain them – but they are immediately derived from this specification and they will be used to define the semantics of joint-actions.

- A *multi-action* is a set of single-agent actions and collaborative actions with consistent preconditions and consistent effects, such that no agent participates in more than one of the actions in this set. Examples of a multi-action would be $\{$*move-left(*a_1*),move-right(*a_2*)*$\}$ or $\{$*move-left(*a_1*),2push(*a_2,a_3,b*)*$\}$.

- We call the actions – single or collaborative – comprising a multi-action, its *components*. For example, the components of $\{$*move-left(*a_1*),2push(*a_2,a_3,b*)*$\}$ are *move-left(*a_1*)* and *2push(*a_2,a_3,b*)*.
- The *primitive elements* of the multi-action are its single-agent action components *and* all single-agent actions that make up any of its collaborative action components. For example, the primitive elements of $\{$*move-left(*a_1*),2push(*a_2,a_3,b*)*$\}$ are *move-left(*a_1*)*, *push(*a_2,b*)*, and *push(*a_3,b*)*.

There is one last step. Not all multi-actions are *well-formed.* A *well-formed* multi-action is one in which interactions have been correctly captured by the collaborative actions in it. Roughly speaking, this means that none of the components of a multi-action can be combined together to form a different collaborative action. That is, if a and a' are single-agent actions in a multi-action, (e.g., two *push* actions by different agents applied to the same object) they can occur in a multi-action only if no collaborative action consisting of a and a' (e.g., a *2push* action), exists. For example, $\{$*move-left(*a_1*),push(*a_2,b*),push(*a_3,b*)*$\}$ is a multi-action, but it is not well-formed because some of its components form a collaborative action, i.e., $\{$*push(*a_2*,b),push(*a_3*,b)*$\}$ form the collaborative action *2push(*a_2,a_3,b*)*. We emphasize that the discussion of well-formedness is intended only to provide a rough intuition, and it will be made precise later on.

We can now specify the semantics of our language: every well-formed multi-action a^m induced by the language is mapped to the joint-action that consists of all the primitive elements of a^m together with a *no-op* for every agent that does not contribute a primitive element to a^m.

Going back to our running example, from the semantics of the multi-action $\{$*move-left(*a_1*),2push(*a_2,a_3,b*)*$\}$, we can obtain the transition function for the joint action that corresponds to $($*move-left(*a_1*),push(*a_2*,b),push(*a_3,b*)*$)$, and from the semantics of the multi-action $\{$*move-left(*a_1*),move-right(*a_2*)*$\}$, we can obtain the transition function for $($*move-left(*a_1*),move-right(*a_2*),no-op*$_3)$.

Summarizing, the essential idea behind our specification is as follows: Multi-actions provide the semantics for joint-actions. Each multi-action is composed of single-agent and collaborative actions that do not interact with each other, i.e., the execution of one does not modify the effects of the other.[2] Therefore, the description of a multi-action can be derived

[2] They may delete each other's preconditions, but we do not care, as we assume true concurrency. This is very much like a classical action deleting one of its preconditions.

from the description of the single and collaborative actions that make it up, and it need not be specified explicitly. Thus, the modeler specifies single and collaborative actions only, where all action interactions are captured by suitable collaborative actions.

5.4.2 Language

A multi-agent planning problem consists of $\langle P, I, g, \Phi, \{A_1, \dots A_n\}, A_c \rangle$, where P is a set of ground primitive propositions, $I \subset P$ is the initial state, $g \subset P$ is the goal condition, Φ is a set of agent names, A_i is a set of *single-agent* actions, and A_c is a set of *collaborative* actions.

A literal l is a, possibly negated, proposition from P, i.e. $l = p$ or $l = \neg p$ for some $p \in P$. Given a set of literals L, let $L^+ = \{p \in P | p \in L\}$ (the positive propositions in L), and let $L^- = \{p \in P | \neg p \in L\}$ (the negative propositions in L). L is well-defined if $L^- \cap L^+ = \emptyset$.

Definition 8 *A* single-agent action *is a tuple $a = \langle symbol, pre(a), eff(a) \rangle$, where symbol is the action name, and pre(a) and eff(a) are well-defined sets of literals. pre$(a)^+$ is the set of positive pre-conditions, pre$(a)^-$ is the set of negative pre-conditions, eff$(a)^+$ is the set of add effects, and eff$(a)^-$ is the set of delete effects.*

Definition 9 *A* collaborative action *is a tuple $a_c = \langle symbol, pre(a_c), eff(a_c), e = \{a_1, \dots, a_k\} \rangle$, where symbol, pre($a_c$) and eff($a_c$) are as above, and e is a set of single-agent action symbols, such that no two action symbols in e belong to the same agent in Φ. We refer to $e(a_c)$ as the* primitive elements *of a_c.*

To simplify the description we use the generic name *action* to refer to either a single-agent action or a collaborative action whenever possible; we will drop the distinction between an action and its symbol; and we write $e(a)$ to denote the primitive elements of an action a. When a is a single-agent action, $e(a) = \{a\}$, and when a is collaborative, $e(a)$ is simply e, the set of single-agent actions in a's definition. We also write $Agt(a)$ to denote the set of agents acting in a: $Agt(a) = \{\phi_i | \exists a_i \in e(a), a_i \in A_i\}$, i.e., agents for whom a contains an element from their action set.

Finally, we note that the input language is not really based on ground actions, but we take a PDDL-*like* view of planning in which actions are instantiated from action templates by replacing parameters with suitable objects. An example of the *2push* action template would be:

```
2push(?agt1,?agt2,?box,?loc1,?loc2)
    precondition: (and at(?agt1,?loc1),at(?agt2,?loc1),
                        at(?box,?loc1),heavy(?box))
```

```
effect: (and (NOT at(?agt1,?loc1)),at(?agt1,?loc2),
               (NOT at(?agt2,?loc1)),at(?agt2,?loc2),
                    (NOT at(?box,?loc1)),at(?box,?loc2))
elements: Push(?agt1,?box,?loc1,?loc2),
          Push(?agt2,?box,?loc1,?loc2)
```

Acting agent(s) typically appear as parameters, as shown above, but nothing prevents the parameters from including other agents as well. Sometimes these agents will be actors, as in a collaborative action, and sometimes they can be passive objects.

For semantic clarity, in what follows we mostly consider the grounded version of the input language.

5.4.3 Model

Our formal semantic model is essentially a transition system with one transition function associated with every joint action, where every joint-action is associated with an n-tuple of single-agent actions. Formally: a *multi-agent planning problem model* $\langle S, A, s_0, G, \varphi, \{\mathscr{A}_i : 1 \leq i \leq n\}\rangle$ is defined as follows: S is a set of states; A is a set of joint actions; $s_0 \in S$ is the initial state, $G \subseteq S$ is the set of goal states, φ is the set of agents, where $|\varphi| = n$ by convention; and $\mathscr{A}_i$ are the single-agent action symbols for agent $\varphi_i \in \varphi$, where $\mathscr{A}_i$ will always contain *no-op*$_i$. Every action $a \in A$ is a pair: a partial function from S to S and a tuple $(a_1, \ldots, a_n)$ of single-agent action symbols, such that $a_i \in \mathscr{A}_i$. As with collaborative actions, $e(a)$ will denote the single-agent action symbols associated with a such that *no-op*$_i$ would be explicitly included in e. We write $a(s)$ to denote the state obtained when applying a in state s. A plan $\pi = a_1, a_2, \ldots, a_k$ is a sequence of joint actions such that $a_k(\cdots(a_1(s_0))) \in G$.

5.4.4 Interpretation

The correspondence between the domain specification and the model is defined as follows: The set of states S corresponds to all possible truth assignments to P. We often equate a state with the list of propositions satisfied in it. Thus, s_0 is the state associated with I. G consists of all states satisfying g. So far, this is identical to classical single-agent planning.

To define the set of joint-actions in the model, we first define the notion of a multi-action, and then associate a joint-action with every *well-formed* multi-action.

Definition 10 *A* multi-action *is a set of actions* $a_m \subseteq A_c \cup (\cup_{i=1}^{n} A_i)$ *such that (1) for every* $a, a' \in a_m : Agt(a) \cap Agt(a') = \emptyset$*; and (2)* $pre(a_m) = \cup_{a \in a_m} pre(a)$ *and* $eff(a_m) = \cup_{a \in a_m} eff(a)$ *are both well defined.*

Condition (1) ensures that no agent will be an actor in more than one action in a_m. Condition (2) ensures that the effects of actions in a_m do not conflict with each other, and that the preconditions of actions do not conflict with each other. This is why we can define: $\text{pre}(a_m) = \cup_{a \in a_m} \text{pre}(a)$ and $\text{eff}(a_m) = \cup_{a \in a_m} \text{eff}(a)$.

Definition 11 *The* components *of a multi-action $a_m = \{a_1, \ldots, a_k\}$ (where $\{a_1, \ldots, a_k\} \subseteq A_c \cup (\cup_{i=1}^{n} A_i)$) are simply $a_1, \ldots, a_k$. We extend the notation e and Agt to multi-actions in the natural manner: $e(a_m) = \cup_{a \in a_m} e(a)$; $Agt(a_m) = \cup_{a \in a_m} Agt(a)$. We refer to members of $e(a_m)$, which are all single-agent actions, as the* primitive elements *of a_m.*

A key concept in the interpretation is that of a *well-defined* multi-action [82]. Let $a_c = \{a_1, \ldots, a_l\}$ and $a'_c = \{a'_1, \ldots, a'_m\}$ be two multi-actions. We say that a_c is *subsumed* by a'_c if: (1) $e(a'_c) = e(a_c)$; (2) for every $a_i \in a_c$ there is some $a'_j \in a'_c$ such that $e(a_i) \subseteq e(a'_j)$; and (3) $m < l$. That is, both multi-actions involve the same set of elements, and moreover, for every member of a_c, there is a member of a'_c that contains all the elements of a_c, and that this containment is strict in at least one case (and hence, $m < l$).

The rationality behind the notion of "well-definedness" is as follows: Because we want the effect of a multi-action to be the union of effects of its member actions, we do not want to allow multi-actions that contain multiple actions that can be combined into a single action, as then the corresponding joint action (defined below) will have an incorrect transition function. As an example, consider our box-pushing domain with 3 agents, and assume that in addition to *push* and *2push*, we also have a three-agent push, *3push*, whose elements are three *push* actions, one for each agent. The multi-action $\{push(a_1,b), push(a_2,b), push(a_3,b)\}$ has three primitive elements, which are also its members. It is subsumed by $\{2push(a_1,a_2,b), push(a_3,b)\}$ which has the same set of elements but only two member actions, and $push(a_1,b), push(a_2,b)$ are mapped into $2push(a_1,a_2,b)$. Both are subsumed by $\{3push(b)\}$. However, neither $\{2push(a_1,a_2,b), push(a_3,b)\}$ subsumes $\{push(a_1,b), 2push(a_2,a_3,b)\}$ nor does the latter subsume the former. Of all these multi-actions, only $\{3push(b)\}$ is well-defined, as the others are subsumed by it. If, on the other hand, we had only *push* and *2push*, but not *3push*, then $\{push(a_1,b), push(a_2,b), push(a_3,b)\}$ would not be well defined because it is subsumed by $\{2push(a_1,a_2,b), push(a_3,b)\}$. The latter, as well as $\{push(a_1,b), 2push(a_2,a_3,b)\}$, would be well defined.

Technically, the definition above is fine. However, the notion of *well-definedness* allows for the formation of "unintuitive" multi-actions. Consider the above example from our box-pushing domain with *three* agents along with *push*, *2push*, and *3push* actions. Note that the multi-action $\{push(a_1,b), push(a_2,b), push(a_3,b)\}$ does not capture the intuition behind the definition of *2push* and *3push*. The latter actions were defined because the effect of *push* is different if other agents are pushing the same box. In that case, we would expect that the only multi-action that contains three *push* primitive elements would be {*3push*}. Similarly, imagine that *3push* was not defined, and only *push* and *2push* were defined. Again, {*push(*a_1,b*),push(*a_2,b*),push(*a_3,b*)*} seems inappropriate, as it contains two single *push* primitive elements, that can be combined into a larger component: *2push*. In fact, for that reason, both {*2push(*a_1,a_2,b*),push(*a_3,b*)*} and {*push(*a_1,b*),2push(*a_2,a_3,b*)*} are problematic. For example, imagine that when only a single agent pushes the box, the box will be pushed, but the agent will become tired. But if two agents push it, the agents are not tired. In that case, although agent a_3 (in the first case) or agent a_1 (in the second case) are really pushing the box alone, the effect of the multi-actions would be that they are tired. Indeed, in this setting, we would want to allow only the actions in which either a single agent performs *push*, such as {*push(*a_1,b*),noop(*a_2*),noop(*a_3*)*} or where two agents are pushing, such as {*2push(*a_1,a_2,b*),noop(*a_3*)*}.

To address this issue we require multi-actions to be *well-formed.*

Definition 12 *A multi-action a_m is* well-formed *if no subset of its primitive elements* $\{ a_{i_1},a_{i_2},\ldots,a_{i_k}\} \subseteq e(a_m)$ *satisfies the following two conditions: (1)* $\{a_{i_1},a_{i_2},\ldots,a_{i_k}\}$ *contains primitive elements from* at least *two actions in a_m. (2) There exists a collaborative action $a_c \in A_c$ such that* $e(a_c) = \{a_{i_1},a_{i_2},\ldots,a_{i_k}\}$.

Going back to our example above: if we have the actions *push* and *2push* then the multi-action: a_m = {*2push(*a_1,a_2,b*),push(*a_3,b*)*} is not well-formed because there is a subset of its primitive elements: {*push(*a_2,b*), push(*a_3,b*)*} that are not part of the same collaborative action in a_m, yet there is a collaborative action *2push(*a_2,a_3,b*)* whose primitive elements are exactly *{push(*a_2,b*), push(*a_3,b*)}*. This implies that if we want to allow three agents to push a box concurrently, we must define an explicit *3push* action. Given a multi-action a, the result of applying a in s, $a(s)$, is well defined if $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. In that case, $a(s) = (s \setminus eff^-(a)) \cup eff^+(a)$.

We can now complete the definition of the interpretation. The set of agent symbols $\varphi = \Phi$. The sets $\mathscr{A}_i$ contain one action symbol for every $a_i \in A_i$, and A contains one joint-action a for every well-formed multi-action a_m. The result of applying joint-action a that corresponds to a_m in s, $a(s)$, is well defined if $pre^+(a_m) \subseteq s$ and $pre^-(a_m) \cap s = \emptyset$. In that case, $a(s) = (s \setminus eff^-(a_m)) \cup eff^+(a_m)$. This, again, is the classical interpretation of an action based on its preconditions and effects. The vector of single-agent symbols associated with a is $e(a_m) \cup \{noop_i : e(a_m) \cap A_i = \emptyset\}$. That is, the multi-action is associated with the joint action in which each agent either executes its single-agent primitive element of a_m (and there is at most one such action), and otherwise, does a *noop*.

Note that there does not necessarily exist a joint-action for every n single-agent action symbols. Intuitively, some combinations of single-agent actions are not well-formed. [3] In a sense, this is similar to the fact that actions in classical planning are not defined on all states, but only on states satisfying their preconditions. However, the following holds and ensures that the joint-actions are well-formed:

Lemma 1 *Let $\{a_1,\dots,a_n\}$ be a joint action. Then, there exists at most one well-formed multi-action a_m such that $e(a_m) = \{a_1,\dots,a_n\}$.*

Proof Suppose to the contrary that there are two different well-formed multi-actions: a_m, a'_m such that $e(a_m) = e(a'_m) = \{a_1,\dots,a_n\}$. Suppose that a_c is some action in a_m that is not in a'_m. If the primitive elements of a_c are a subset of some action a'_c in a'_m then a_m is not well-formed because we can combine $e(a_c)$ with a few other elements to get $e(a'_c)$. Otherwise, the primitive elements of a_c are a counter-example to the well-formedness of a'_m. □

Note that from here onwards, our primary focus will be on the notion of well-formedness. For more details on how exactly we handle our initially proposed, a weaker version of well-formedness, i.e., *well-definedness*, we refer the reader to Section 5.8.

5.4.5 Object Cardinality Constraints

CJR's object cardinality constraints constrain the set of legal joint actions. Their intuition is very appealing – actions typically interact through joint objects, and the number of agents that can manipulate a set of objects concurrently is often constrained. For example, there is a maximal number of agents that can cross a bridge at one time, or there is a minimal number of agents that can use a boat at the same time – e.g., because at least two are required to sail the boat. According to the semantics of CJR, a legal joint action is any combination of single-agent actions that satisfies the cardinality constraints, and the union of the preconditions and the union of the effects of its contained actions is well defined. In the service of simplicity, we do not discuss these constraints formally. It is not hard to use CJR's ideas to extend our description to support them, and our implementation, which builds on their code, does this. See [23] for more details. Perhaps, we also briefly discuss their approach and the resulting planner in Chapter 2, in Section 2.1.2.

[3] Alternatively, define all joint-actions for which no well-formed multi-action exists to have *false* as their preconditions.

5.5 Planning With Multi-Actions

We now consider planning with multi-actions, separating the treatment into two cases: multi actions whose member actions do not interfere, i.e., no action adds or deletes a precondition of another action (recall – by definition, the effects themselves must be consistent), and the more general case, referred to as multi-actions with pre/eff interactions.

5.5.1 Non-Interfering Actions

If multi-actions containing interfering actions are not allowed, then all allowed interactions are already captured by the use of collaborative actions. Hence, the only benefit of performing them jointly is make-span reduction. That is, the set of states reachable with multi-actions and with (single-agent and collaborative) actions is identical. To see this, consider a multi-action consisting of actions $a_1,\ldots,a_k$. As no action deletes the precondition of the other, assuming the preconditions of all actions hold initially, they remains true following the execution of any subset of these actions. Hence, they can be executed in sequence. Since no action deletes the effects of another action, the effect of executing them in sequence is identical to that of executing them concurrently. Thus, we can use any single-agent classical planning algorithm to solve the problem by combining single-agent and collaborative actions together to obtain a single-agent planning algorithm in which the agents are simply objects. Once a plan is obtained, a parallelization algorithm, such as CJR's, can be used to reduce make-span by concurrently executing non-interfering components.

Object cardinality constraints can be supported similarly. Maximum constraints can be enforced directly by the parallelization algorithm without modifying the domain model or the planning algorithm. Minimum cardinality constraint can be compiled away as follows: replace the single-agent actions with a collaborative action (or multiple collaborative actions in some cases) involving the minimal number of agents. For example, if exactly two people are required to cross a bridge, we remove the *cross* action, replacing it with a *2cross* collaborative action consisting of two *cross* actions. If two or more people can cross the bridge, we remove *cross* and add a both *2cross* and *3cross* actions: any number of agents > 2 can cross the bridge by sequencing multiple *2cross* and *3cross* actions without losing completeness, because if the actions do not interfere, there is no difference between executing k *cross* actions followed by m *cross* actions versus doing all $k+m$ actions at the same time.

To summarize, we can address the multi-agent model of CJR by introducing collaborative actions that capture minimality constraints, and use the original domain with these added actions + post-processing. At most two action schema with an arity of 3 are required – leading to a number of ground actions cubic in the number of agents.

5.5.2 Multi-Actions with Pre/Eff Interactions

The above compilation scheme may become both unsound and incomplete when we allow multi-actions that contain actions that delete or add preconditions of other actions. Such action interactions seem natural when we consider true concurrency. For example, there is no reason we would want to exclude two agents from concurrently pushing a box, even though each push action deletes the preconditions of the other by changing the location of the box. Note that this is an issue regardless of collaborative actions. For example, if sailing a boat changes its location, then multiple agents cannot sail the boat if we do not allow multi-actions that destroy each others' preconditions. In some cases, the goal may be reachable only if we allow multi-actions of this kind, and unreachable without them. For example, due to limited resources, the boat might be able to sail in one direction only, in which case, we must ensure all agents that need to take it sail at the same time. In that case, post-processing action sequences will not suffice, and we need to actively generate well-formed multi-actions. This requires a non-trivial compilation scheme.

Before we present this compilation, we discuss two issues. The first issue is the impact of allowing interfering action and minimality constraints on the definition of well-formed multi-actions. To see this, consider the example of the *sail* action, and suppose that we require at least two agents to sail the boat. To handle the minimality constraint, we added *2sail* and *3sail*. If these actions do not interfere with each other, we can execute them in sequence. If they interfere, then we need to allow multi-actions that contain multiple instances of that action. For example, to allow four agents to sail on the same boat, we could insert two *2sail* actions by different agents: *2sail*(a_1,a_2*,b,o,d)* and *2sail*(a_3,a_4*,b,o,d)*. This, however, would make the action not well-formed because from its elements *sail*(a_1*,b,o,d)* and *sail*(a_3*,b,o,d)* we can create the collaborative action *2sail*(a_1,a_3*,b,o,d)*, which is not part of the multi-actions.

Note that this issue arises specifically from our approach for handling minimality constraints. If we do not add the *2sail* and *3sail* actions, and remain with just *sail*, then the multi-action {*sail*(a_1*,b,o,d)*, *sail*(a_2*,b,o,d)*, *sail*(a_3*,b,o,d)*, *sail*(a_4*,b,o,d)*} is well-formed. A simple way to address it is to simply support minimality constraints directly.[4]

A second issue to consider if we allow actions that delete preconditions of each other, is the subtle semantic (rather than syntactic) issue of when do two actions interfere, or even conflict. If we allow a multi-action containing *sail(a1, boat, origin, destination)* and *sail(a2, boat, origin, destination)*, why should we not allow a multi-action containing *sail(a1, boat, origin, destination1)* and *sail(a2, boat, origin, destination2)*? Intuitively, we view

[4] It is also possible to slightly modify the definition of well-formed actions to allow for more flexibility as long as Lemma 1 remains true. This can be done when the elements are non-interacting.

the effects: *at(boat, destination1)* and *at(boat, destination2)* as inconsistent. While this would be clear with a multi-valued formulation of the problem, it is not obvious in the boolean case, as the two propositions are logically consistent. In single-agent planning such situations (e.g., *on(a,b), on(a,c)*) do not arise when the initial state is consistent and actions are formulated properly. But as evident, this is no longer true in the multi-agent case. Thus, for this work we assume that additional declarative information about when actions conflict is provided. We will use this information to rule out actions with inconsistent effects. In our implementation, we handle this by adding additional cardinality constraints on concurrent actions. For example, we constrain the number of possible destinations of sail actions for the same object to 1.

```
( define  (domain box−pushing)
(: requirements  : typing )
(: types  agent  box  bridge  location )
(: predicates
    (box−at ?b − box ?loc − location )
    ( intact  ?b − box)
    ( at  ?a − agent  ?loc − location )
    (has−bridge  ?b − bridge  ?loc1 − location  ?loc2 − location )
    ( tired  ?a − agent))
(: action  cross
    : parameters    (?a1 − agent  ?b − bridge  ?loc1 − location  ?loc2 − location )
    : precondition  (and ( at  ?a1  ?loc1 )  (has−bridge  ?b  ?loc1  ?loc2))
    : effect        (and ( at  ?a1  ?loc2)  (not  (has−bridge  ?b  ?loc1  ?loc2))
                         (not  (has−bridge  ?b  ?loc2  ?loc1))  (not  ( at  ?a1  ?loc1))))
(: action  push
    : parameters    (?a1 − agent  ?b − box ?loc1 − location  ?loc2 − location )
    : precondition  (and ( at  ?a1  ?loc1 )  (not  ( tired  ?a1))  (box−at ?b  ?loc1 ))
    : effect        (and ( tired  ?a1)  (not  ( intact  ?b))))
(: action  2push
    : parameters    (?a1 − agent  ?a2 − agent  ?b − box ?loc1 − location  ?loc2 − location )
    : precondition  (and ( at  ?a1  ?loc1 )  ( at  ?a2  ?loc1 )  (box−at ?b  ?loc1 )
                         (not  ( tired  ?a1))  (not  ( tired  ?a2)))
    : effect    (and ( at  ?a1  ?loc2)  ( at  ?a2  ?loc2)  (not  ( at  ?a1  ?loc1 ))
                 (not  ( at  ?a2  ?loc1 ))  (box−at ?b  ?loc2)  ( intact  ?b)
                 (not  (box−at ?b  ?loc1 )))
    :element  (and  (push  ?a1  ?b  ?loc1  ?loc2)  (push  ?a2  ?b  ?loc1  ?loc2))))
```

Listing 5.1 The Box-Pushing example domain

5.5.3 The compilation scheme

We now explain how to compile multi-agent planning problems with collaborative actions into single-agent planning problems. This part can be viewed as extending the compilation of CJR to (1) Properly address pre/eff conflicts; (2) Support collaborative actions; and (3) Ensure that multi-actions are well-formed. Their basic idea was to represent a joint-action using a sequence of actions, demarcated by a special *start* and *end* actions, that has the same effects as the joint-action. As noted, when actions in a multi-action do not interfere, this is relatively straightforward. Our compilation alters the action description so that such serialization can still work in the more general case.

The description below strives for simplicity, rather than economy, and ignores the handling of cardinality constraints, which is identical to CJR. Furthermore, CJR assume that every multi-agent action manipulates a particular set of objects. This has practical benefits, and we exploit this idea in our implementation, but to simplify the presentation, we ignore it, here.

To help clarify the steps, we will be using a very simple running example. The domain consists of agents, boxes, bridges, and locations. In this domain, boxes need to be pushed to their goal locations, an agent can move between connected locations by crossing bridge, but a bridge collapses after it is crossed once, so it cannot be reused. The domain has *agent*, *box*, *bridge* and *location* as the object types. The single-agent actions are: *cross* and *push*, and there is one collaborative action: *2push*. Their description appears in Listing 5.1 (above).

Given a specification of a multi-agent planning problem $\langle P, I, g, \Phi, \{A_1, \ldots A_n\}, A_c \rangle$, we generate the classical planning problem $\langle P_{Cl}, A_{Cl}, I_{Cl}, g_{Cl} \rangle$. In what follows let $A = A_1 \cup \cdots \cup A_n \cup A_c$.

- $P_{Cl} = P \cup P_{act} \cup P_{neg} \cup P_{pos} \cup P_{taken} \cup \{in\}$, where $P_{act} = \{p_a : a \in A\}$; $P_{neg} = \{Neg\text{-}p | p \in P\}$; $P_{pos} = \{Pos\text{-}p | p \in P\}$; and $P_{taken} = \{taken_i | \phi_i \in \Phi\}$. Intuitively, $p_a \in P_{act}$ tells us that the current multi-action contains a; P_{neg} and P_{pos} keep track of changes caused by components of the multi-action; P_{taken} keeps track of which agents are involved in the current multi-action; and *in* tells us that the current multi-action has not ended yet.

  For our running example, the original propositions are grounded instances of *(box-at ?b - box ?loc - location), (intact ?b - box), (at ?a - agent ?loc - location), (has-bridge ?b - bridge ?loc1 - location ?loc2 - location),(tired ?a - agent).* The compiled domain will contain, in addition: P_{act}, which contains the ground instances of *(p-cross ?a1 - agent ?br - bridge ?loc1 - location ?loc2 - location) (p-push ?a1 - agent ?b - box ?loc1 - location ?loc2 - location) (p-2push ?a1 ?a2 - agent ?b - box ?loc1 - location ?loc2 - location)*; P_{pos}, which contains grounded instances of *(pos-box-at ?b - box ?loc - location) (pos-intact ?box - box) (pos-at ?agt - agent ?loc - location) (pos-has-bridge ?br - bridge ?loc1 - location ?loc2 - location) (pos-tired ?agt - agent)*, and analogously for P_{neg}; And finally, P_{taken}, which contains the ground instances of *(taken ?agt - agent).*

- $A_{Cl} = \{a' | a \in A\} \cup \{a_{start}, a_{end}\}$, where each $a' \in A_{Cl}$ is a modification of some $a \in A$, a_{start} makes *in* true, and a_{end} marks the end of a multi-action and does some book keeping and updates.

- $I_{Cl} = I$

- $g_{Cl} = g \wedge \neg in$

We explain the role of the additional variables and the changes in the actions below.

1. a_{start} : pre(a_{start}) = $\{\neg in\}$, eff(a_{start}) = $\{in\}$. Together with a_{end} (defined below) it marks the start and end of multi-action. Thus, the multi-actions $\{a_1, a_2, a_3\}$ will appear in the compiled plan as $a_{start}, a'_1, a'_2, a'_3, a_{end}$.

   For example, the solution plan to our running example with two agents that must push a box on the other side of the bridge while keeping it intact, consists of two multi-actions: In the first, both agent cross the bridge, i,e., they perform *cross(a1, br1, loc1, loc2), cross(a2, br1, loc1, loc2)*, and in the second, they perform the collaborative *2push(a1, a2, box1, loc2, loc3)*. The solution to the compiled problem is a standard sequential plan that encodes this plan and takes the form: $\langle a_{start}$, *cross(a1, br1, loc1, loc2), cross(a2, br1, loc1, loc2)*, $a_{end}\rangle$, $\langle a_{start}$, *2push(a1, a2, box1, loc2, loc3)*, $a_{end}\rangle$.

2. Every action $a \in A$ is modified as follows:

   (a) Every effect p and $\neg p$ is replaced by *Pos-p* and *Neg-p*, respectively. This is used to allow actions in a multi-action that destroy each other's preconditions. In the compiled problem, instead of destroying p, we add *Neg-p*.

   For example, in the *cross* schema we replace the original effects *(at ?a1 ?loc2)* by *(pos-at ?a1 ?loc2)*, *(not (has-bridge ?b ?loc1 ?loc2))* by *(neg-has-bridge ?b ?loc1 ?loc2)*, etc.

   As we will see, a_{end} will update the value of P at the end of the multi-action based on the value of these propositions.

   (b) We add p_a to its effects. In addition, if a is a single-agent action, we add p_a to the effect of every collaborative action a_c such that $a \in e(a_c)$.

   In our running example, we add *(p-push ?a1 ?box ?l1 ?l2)* as an effect of *push*. This same effect is also added to the action *2push* because *push* $\in$ *e(2push)*.

   (c) For every action a', if the effects of a and a' are inconsistent, we add $\neg p_{a'}$ as a precondition to a. Recall our earlier discussion of inconsistent effect – the user may need to explicitly express the fact that certain effects are inconsistent.

   In the example domain, *push* and *2push* have conflicting effects on the object *box*, i.e., *(intact ?box)*. Therefore, those two actions cannot appear simultaneously in a multi-action. The compilation scheme adds *(not (p-push ?a1 ?box ?l1 ?l2))* in the preconditions of *2push* and *(not (p-2push ?a1 ?a2 ?box ?l1 ?l2))* in the preconditions of *push*.[5]

(d) To ensure well-formedness, we add the following precondition to action a:

$$\bigwedge_{\{a_c \in A_c : e(a) \cap e(a_c) \neq \emptyset, a \neq a_c\}} \neg (\bigwedge_{a_i \in e(a_c) \setminus e(a)} p_{a_i})$$

This condition prevents adding a new action to the multi-action such that some of the primitive elements of this new action, together with the primitive elements of previously added actions, are precisely the primitive elements of some collaborative action.

Consider our running example with three agents: a_1, a_2, a_3. The multi-action ⟨*push(a1, box, loc1, loc2), 2push(a2, a3, box, loc1, loc2)*⟩ has the following elements: ⟨*push(a1, box, loc1, loc2), push(a2, box, loc1, loc2), push(a3, box, loc1, loc2)*⟩. Combining the first two elements of this joint-action forms another multi-action, i.e., ⟨*2push(a1, a2, box, loc1, loc2), push(a3, box, loc1, loc2)*⟩. Therefore this multi-action is not well-formed. The compilation approach must restrict such combination of actions explicitly. The updated schemas of *push* and *2push* in the compiled example problem tackle this issue explicitly.

(e) To ensure no agent acts more than once in a multi-action, an additional effect of agent ϕ_i's action is $taken_i$ and an additional precondition is $\neg taken_i$. Thus, in our example, a precondition of *cross(a1, br1, loc1, loc2)* is *(not(taken ?a1))* and an effect is *(taken ?a1)*.

3. a_{end} has the effect $\neg in$, to denote that a multi-action has ended; the conditional effect $Pos\text{-}p \rightarrow p$ and $Neg\text{-}p \rightarrow \neg p$ for every proposition p, to update the state with the effects of all primitive elements of the multi-action; and the effect $\neg p_a$ for every $a \in A$ and $\neg taken_i$ for every agent, to reset these propositions; and the effect $\neg Pos\text{-}p$ and $\neg Neg\text{-}p$ for every $p \in P$, to reset these variables.

   The entire a_{end} action for our example domain appears in Listing 5.2.

At the end of this chapter, we can find the entire compiled domain in Listing 5.3 . We stress again, that a more economical representation is possible, where multi-actions consider a fixed set of objects only. On the one hand, this requires multiple copies of various propositions and actions, but each action is much smaller. In particular, the *end* action for a

[5] *2push* and *push* cannot appear together also because they would cause the multi-action to be not well-formed.

```
(: action  a−end
    : parameters ()
    : precondition (and  (in))
    : effect  (and  (not  (in))
                ( forall  (?agt  −  agent)(not  (taken  ?agt)))
                ( forall  (?b  −  box ?loc  −  location )
                        (and  (when (pos−box−at ?b ?loc)(box−at ?b ?loc))
                              (when (neg−box−at ?b ?loc)(not  (box−at ?b ?loc))))
                ( forall  (?a  −  agent  ?l  −  location )
                        (and  (when (pos−at  ?a  ?l)(at  ?a  ?l))
                              (when (neg−at  ?a  ?l)(not  (at  ?a  ?l)))))
                ( forall  (?a  −  agent)
                        (and  (when (pos−tired  ?a)( tired  ?a))
                              (when (neg−tired  ?a)(not  ( tired  ?a)))))
                ( forall  (?b  −  bridge  ?loc1  −  location  ?loc2  −  location )
                        (and  (when (pos−has−bridge ?b  ?loc1  ?loc2)(has−bridge  ?b  ?loc1  ?loc2))
                              (when (neg−has−bridge ?b  ?loc1  ?loc2)(not  (has−bridge  ?b  ?loc1  ?loc2)))))
                ( forall  (?b  −  box)
                        (and  (when (pos−intact  ?b)( intact  ?b))
                              (when (neg−intact  ?b)(not  ( intact  ?b)))))
                ( forall  (?a1  −  agent ?b  −  bridge  ?loc1  −  location  ?loc2  −  location )
                        (and  (when (p−cross  ?a1  ?b  ?loc1  ?loc2)(not  (p−cross  ?a1  ?b  ?loc1  ?loc2)))))
                ( forall  (?a1  −  agent ?b  −  box  ?loc1  −  location  ?loc2  −  location )
                        (and  (when (p−push ?a1  ?b  ?loc1  ?loc2)(not  (p−push ?a1  ?b  ?loc1  ?loc2)))))
                ( forall  (?a1  −  agent  ?a2  −  agent  ?b  −  box ?loc1  −  location  ?loc2  −  location )
                        (and  (when (p−2push ?a1  ?a2  ?b  ?loc1  ?loc2)(not  (p−2push ?a1  ?a2  ?b  ?loc1  ?loc2)))))))
```

Listing 5.2 Compiled action schema for a_{end}

specific object will have to update only propositions relevant to this set of objects and actions that manipulate this set of objects.

5.5.4 Formal Properties

The main property of our compilation is that it is sound and complete:

Lemma 2 *Let* $\Pi_S = \langle P, I, g, \Phi, \{A_1, \dots A_n\}, A_c\rangle$ *be a specification of a multi-agent planning problem. Let* $\Pi_M = \langle S, A, s_0, G, \Phi, \{A_i : 1 \le i \le n\}\rangle$ *be the model it specifies. Let* $\Pi_{Cl} = \langle P_{Cl}, A_{Cl}, I_{Cl}, g_{Cl}\rangle$ *be the classical planning problem into which* Π_S *is compiled.* Π_M *is solvable iff* Π_{Cl} *is solvable.*

Proof Let π_M be a solution to Π_M. Let π_{Cl} be a classical plan that corresponds to it: each multi-action in $a_m \in \pi_M$ is replaced by $a_{start}, a'_1, \dots, a'_k, a_{end}$, where $a_1, \dots, a_k$ are the members of a_m (ordered arbitrarily). We claim that π_{Cl} is a solution to Π_{Cl}. By definition of the interpretation, s_0 assigns *true* exactly to all propositions in I, and the true propositions in I_{Cl} are the same as in I. In particular, all added propositions are initially *false*. First, notice that after a_{end} is executed, we are in a similar situation: all added propositions have the value *false*. Thus, it is enough to show that the first step of π_{Cl} and of π_M yield an identical assignment to P. Let a_m be the first action in Π_M. Given the definition of the interpretation and Lemma 1, we can treat it as a multi-action. We know that the union of preconditions and the union of effects of its primitive elements are well defined, and the union of their preconditions is satisfied in I_{Cl} and in all the following states, until a_{end} is executed. This is because until a_{end} is executed, none of the propositions in P changes its value. The other preconditions of actions $a'_1,\ldots,a'_k$ must also be satisfied because a_m is well-formed. One precondition is $\neg taken_i$, where agent ϕ_i is (one of) the actors in that action. Since it is not an actor in any other action, $taken_i$ will be *false* until ϕ_i's action is executed. The second type of preconditions is added in $2(d)$. We claim that because all actions are well-formed, it is satisfied. Imagine such a precondition of a'_i is not satisfied. This implies that for some action a_c, the elements of a'_i together with the elements of actions $a_1,\ldots,a_{i-1}$ contain the elements of a_c. In that case, a_m is not well formed.

Next, let π_{Cl} be a solution to Π_{Cl}. π_{Cl} must be a sequence of action sequences of the form $a_{start}, a'_1,\ldots,a'_k, a_{end}$. a_{start} is the only action executable when *in* is *false*. a_{end} is the only action that makes *in false*, and $\neg in$ is part of g_{Cl}. As noted above, all added propositions are *false* initially and after a_{end} is executed. So it remains to show that $\{a'_1,\ldots,a'_k\}$ corresponds to a well-formed multi-action. Because the propositions in P change value only in a_{end}, the union of preconditions of $a'_1,\ldots,a'_k$ must be satisfied in the state where a_{start} is applied, so the union is well-defined. For the effect not to be well-defined, two elements of this action must have conflicting effects, but if a_i conflicts with a_j, then $\neg p_{a_i}$ is a precondition of a_j, and they cannot occur together in this sequence. No agent can act twice, because of the $taken_i$ propositions. Finally, if some combination of elements appearing in $a'_1,\ldots,a'_k$ corresponds to a different collaborative action, the last action in this combination could not be applied because of constraint $2(d)$. □

Lemma 3 *The size of the classical encoding is worst-case cubic in the size of the multi-agent planning problem.*

Proof Clearly, the number of propositions is linear in the number of original propositions and actions, whereas the number of actions increases by two. The actions, however, are modified to include additional effects. In the worst case, $2|A|$ new effects are added to each action due to $2(b)$ and $2(c)$. Condition $2(d)$ adds a precondition whose size is bounded by the number of actions times the maximal size (number of elements) of a collaborative action. Potentially, this needs to be done for a linear number of actions, leading to a quadratic increase in size per action, and a cubic increase overall. In practice, the impact of $2(b-d)$ is likely to be linear. For example, $2(d)$ considers only actions with shared elements. Finally, a_{end} has linear size. □

5.6 Adding Privacy

Privacy Preserving Planning (PPP) [58] supports agents that wish to plan collaboratively without revealing private information about their local state, their private actions, and their cost. For example, producers cooperating on a joint product will want to expose the capabilities they can contribute to the project, without necessarily revealing the identity of their suppliers and employees, their internal processes, and their inventory level. PPP algorithms are able to compute a joint-plan in a distributed manner without revealing private information. We now explain how to modify our specification and compilation technique to support PPP with interacting actions.

The input to a PPP problem differs from that of a centralized multi-agent planning problem: Each agent has a separate domain specification that contains a description of its actions. This specification also differentiates between *private* and *public* propositions and between *private* and *public* actions. A proposition may be private to an agent only if it does not appear in the description of actions of other agents. An action can be private only if its description contains private propositions only. Public actions may contain both private and public propositions. Their *public projection* is obtained by removing all private propositions from their description.

5.6.1 Modifying the Representation

In PPP, the domain description of each agent contains: a complete description of all its actions and the public projection of the public actions of other agents; together with public propositions and propositions private to the agent. We extend this description with collaborative actions.

A collaborative action is public by definition, as it involves multiple agents. For the same reason, we will also assume that single agent actions that interact with actions of other agents are public, too. That is, single agent actions that appear as elements of collaborative actions. By definition, their effect depends on the actions of other agents, so in a collaborative setting, we would expect this information to be public. This implies that propositions of the form p_a that are added in our compilation are also public. Note that we need to add a p_a proposition for an action a only when a is public.

Although collaborative actions are public, some of their preconditions or effects could be private to one of the agents. For example, the action *2push*(a_1,a_2,b) may have the precondition *healthy*(a_1) private to a_1, and a private effect *tired*(a_1). The description of *2push* in a_2's domain description will not include these preconditions and effects, i.e., they are projected out. Notice that while the specification is now distributed among n agents, the semantics remains the same.

5.6.2 Privacy Preserving Planning with Collaborative Actions

To perform privacy preserving planning with collaborative actions, the first step is to apply our compilation scheme to the public part of the domain. All added propositions and actions are public. Private actions and propositions need not be changed, and the private preconditions and effects of actions need not be modified. In particular, if we allow interfering actions, then we must add a_{start}, a_{end}, as is required in the compilation for this case, and these actions can be assigned to arbitrary agents. Notice that these actions manipulate public propositions only, so no privacy is lost by assigning them to a particular agent.

The next step is to use an existing PPP algorithm to solve the resulting problem. However, existing PPP algorithms are distributed, and this raises the question of when to insert a collaborative action into the plan. One agent cannot commit to a collaborative action in the name of another agent because it does not know if the private preconditions of that agent hold. To address this, Brafman and Zoran (2014) added a special message between the agents to address this. We believe that splitting collaborative actions as follows is a simpler solution which allows us to use any existing PPP planner.

1. Add the precondition $\neg$*in-joint* to all existing public non-collaborative actions;
2. For each collaborative action a_m involving k agents:
   (a) Separate a_m into k actions $a_m^1, \ldots, a_m^k$, where a_m^i is obtained from a_m by removing the private preconditions and effects of agents other than ϕ_i. The parameters of each new action that do not appear in its preconditions or effects are removed.
   (b) Add to a_m^1 precondition $\neg$*in-joint* and effect *in-joint*.
   (c) Add to a_m^k the effect $\neg$*in-joint.*
   (d) For all $i < k$, add to a_m^i the effect *next-*a_m^{i+1}.
   (e) For all $i > 1$, add to a_m^i the precondition *next-*a_m^i and the effect $\neg$*next-*a_m^i.

For example, we split *2push(*a_1*,*a_2*,b)* into *2push*$_1$*(*a_1*,*a_2*,b)* and *2push*$_2$*(*a_1*,*a_2*,b).* If a_2 is not mentioned in the description of *2push*$_1$ and a_1 is not mentioned in the description of *2push*$_2$, we obtain *2push*$_1$*(*a_1*,b)* and *2push*$_2$*(*a_2*,b).* Thus, the first pushing agent need not commit to the identity of the second pushing agent. *2push*$_1$ will have $\neg$*in-joint* as a precondition and *in-joint* $\wedge$ *next-*2*push*$_2$ as an effect. *2push*$_2$ will have *next-*2*push*$_2$ as a precondition and $\neg$*in-joint* $\wedge\neg$*next-2push*$_2$ as an effect.

There remains one subtle issue when a MAFS-based algorithms [58] is used. In MAFS, agents must end every sequence of private actions with a public action. Imagine that we attempt to insert into a plan, a collaborative action such as *2push(a_1,a_2,b)* that has two preconditions: p_1 is private to a_1 and p_2 is private to a_2, and these preconditions are initially *false*. Suppose that the first agent uses private action a_{p_1} to achieve p_1 and then applies *push*$_1$*(a_1,b)* (which is public and was split as described above). At this point a_2 cannot apply *push*$_2$*(a_2,b)* because p_2 does not hold. Suppose a_{p_2} is private and achieves p_2. We must allow a_2 to perform a_{p_2} before applying *push*$_2$*(a_2,b)*. This is indeed possible because $\neg$*in-joint* is not a precondition of private actions. However, the actions now appear in the order: a_{p_1}, *push*$_1$*(a_1,b)*, a_{p_2}, *push*$_2$*(a_2,b)*. But collaborative actions must be executed in the same time, so we must push back all intermediate private actions to before the first part of the collaborative action, to obtain a_{p_1},a_{p_2}, *push*$_1$*(a_1, b)*, *push*$_2$*(a_2,b)*. Because private actions of one agent do not interact or interfere with actions of other agents, such re-ordering does not impact the result of the plan, and is correct.

5.7 Empirical Evaluation

We now evaluate the scalability of our compilation approach for the regular MAP and for PPP. Existing domains do not incorporate fully interacting actions, but only limited aspects of them. Thus, we defined a new set of domains with interacting actions, and for each domain, we generated a centralized version and a distributed version with private elements.

For each centralized domain, we compared three different compilation schemes. In the first two variants, we adapt the strategy of CJR in which all single-agent actions within each joint-action deal with a fixed set of objects. The code for generating both compilations is built upon the software of Furelos-Blanco, Frances, and Jonsson (2019) and exploits their encoding of object constraints. The third variant removes this restriction – i.e., joint-actions can contain actions manipulating different object sets. The code for this variant is not based on Furelos-Blanco, Frances, and Jonsson (2019). More specifically: the first variant (Rep1 [82]) is based on a formalism we proposed earlier which used a notion weaker than well-formedness, called *well-defined* (see Section 5.8 for more details on this including how the general compilation scheme described in Section 5.5.3 differs to ensure *well-definedness*). As noted above, it allows only multi-actions, all of whose actions manipulate a fixed set of objects. This compilation scheme also incorporates the object cardinality constraints of CJR. The second variant is identical to *Rep1*, except that it enforces the requirement for *well-formed* multi-actions instead of *well-defined*. While this compilation includes the cardinality constraints mechanism of CJR, as well, it requires us to add additional collaborative actions, as we now explain.

A well-defined multi-action can contain multiple instantiations of similar collaborative actions (e.g., *2push(a1,a2,b)* and *2push(a3,a4,b)*), whereas such multi-actions are not well-formed. Thus, if we want to allow between 2 to 4 simultaneous rowers for a boat, in the case of well-defined multi-actions, we can make do with the *row* and *2row* actions, as they can be combined. But if we require multi-actions to be well-formed, we must specifically define *2row*, *3row*, and *4row*.

The third variant (Rep3) is based on the formalism we define in this chapter. We do not enforce any constraints on the content of a multi-action, and we require multi-actions to be *well-formed*. In each domain a general PDDL-like representation is used, e.g, the action *2push* has two agent parameters and one box parameter.

5.7.1 Domains

We used four PDDL-like domains in our experiments of which two are new, and the other two modify existing domains.

Domain	Ins (#agents)	Size	Rep 1: Well-Defined			Rep 2: Well-Formed			Rep 3: Well-Formed (*general*)		
			Length	Makespan	Time (s)	Length	Makespan	Time (s)	Length	Makespan	Time (s)
Box-Pushing	P01 (3)	8	15	11	0.03	13	9	0.06	14	10	0.04
	P02 (4)	12	38	25	1.04	22	14	0.54	26	16	0.78
	P03 (4)	12	43	21	3.11	48	24	5.83	60	36	3.31
	P04 (5)	16	99	62	13.93	106	50	54.55	105	68	80.25
	P05 (5)	18	100	56	62.32	103	68	102.16	96	60	33.28
	P06 (7)	25	156	97	356.20	212	77	1102.04	-	-	-
	P07 (7)	28	223	149	1459.80	-	-	-	-	-	-
Maze	P01 (3)	49	12	12	0.30	14	12	0.44	12	12	0.01
	P02 (4)	50	34	30	108.06	34	27	560.75	50	41	0.37
	P03 (5)	51	37	33	396.61	33	26	1151.86	59	48	10.60
	P04 (4)	52	35	28	1587.78	31	23	485.92	50	34	25.36
	P05 (6)	55	-	-	-	-	-	-	104	73	368.41
	P06 (6)	55	-	-	-	-	-	-	125	97	1412.90
Table-Movers	P01 (3)	9	23	15	0.52	36	20	1.88	28	20	0.08
	P02 (4)	11	29	20	6.10	40	22	26.30	34	25	0.48
	P03 (5)	13	68	53	312.20	-	-	-	68	53	151.40
	P04 (5)	15	58	42	885.10	84	58	1069.88	76	58	16.41
	P05 (6)	14	-	-	-	-	-	-	51	38	995.73
Apartment-Movers	P01 (3)	14	44	32	1.32	43	29	2.82	26	18	0.04
	P02 (4)	24	88	72	17.18	120	96	36.16	65	49	0.84
	P03 (5)	25	98	82	24.76	-	-	-	62	46	1.51
	P04 (4)	26	103	83	27.91	132	96	139.84	73	53	1.20
	P05 (5)	32	140	118	1204.09	-	-	-	96	74	54.27
	P06 (5)	36	-	-	-	-	-	-	98	74	108.07
	P07 (6)	37	143	119	**1965.63**	-	-	-	107	83	170.14
	P08 (8)	38	-	-	-	-	-	-	-	-	-

Table 5.1 Comparison of different interpretations: Well-defined is the interpretation we used in [82]. Well-formed is the new interpretation proposed in this work. The column *well-formed* (the general approach) shows the results when the planner is not bound to add actions in the current multi-action that manipulates only one subset of objects. "-" represents no solution found given the time and memory limits.

Maze

This domain is a modified version of the Maze domain used by CJR. Based on their description, it covers situations in which multiple actions *must* or *cannot* be performed together. It also includes resources that cannot be utilized more than once in the planning.

The domain consists of a 2D grid, in which agents start from their initial locations and each agent has a dedicated goal location. Two adjacent cells in this 2D grid are connected, and one cell can be reached from the other via moving through a *door*, crossing a *bridge*, or rowing a *boat*. Only one agent can pass through a door at a time, while several agents can cross a bridge simultaneously. However, this bridge collapses after its first use. In our description, a boat can be rowed by exactly two agents (and thus, following our explanation above, we introduced a collaborative action *2row*. In addition to that, some doors may be closed initially and they will have an associated *switch* at some arbitrary location in this grid. This switch must be pushed to open the door.

In the privacy preserving planning settings, the location of an agent in the grid is private to that agent.

TableMovers

The Table-Movers domain consists of a number of tables and rooms, and agents can move between connected rooms, only. Tables are placed at their initial locations, and agents are supposed to move the table to their dedicated goal locations. Each table has some fragile items on top of it, and these objects must remain *intact*. The primitive single-agent actions in this domain are: *move-agent*, *charge-agent*, *move-table*, *lift-table*, *drop-table*. The collaborative actions are: *2move-table*, *2lift-table*, *2drop-table*, where each one of is a composition of the corresponding two primitive single-agent actions.

If an agent lifts/drops a table alone, objects on top of this table fall and the table is no longer intact. But the collaborative actions *2lift-table* and *2drop-table*, respectively, lift and drop the table simultaneously, keeping it intact in the process. Agents can move a table only if they are charged. Charging points are available only in some rooms. In the PPP settings, the location of an agent and its charging status are private information.

BoxPushing

This domain is a modified version of the box-pushing domain described in [19]. It consists of agents, boxes and locations. Each box is kept at some initial location, and the goal is to move all boxes to their goal locations. The primitive single-agent actions are: *move-agent* and *push-box*, while the only collaborative action is: *2push-box*, which is a composition of two primitive *push* actions. The single-agent *push* moves the box to a connected location. An agent pushing a box alone will hurt her back and cannot perform additional *push* actions. But when using the collaborative action, the box moves without this side effect. The location of an agent and whether her back hurts are private to that agent.

ApartmentMovers

This domain is based on the classical *Depot* domain. It consists of locations and trucks. Trucks are used to move furniture and electronic items from one location (Apartment A) to another location (Apartment B), if they share a connected path. Electronic items are fragile and, therefore, must be packed in a carton-box before they are moved. There is no limit on the number of items that can be packed in a carton-box.

The primitive actions in this domain are: *consume*, *move-agent*, *drive-truck*, *load-carton*, *unload-carton*, *load-furniture*, *unload-furniture*, *pack-appliance* and *unpack-appliance*. Like the previous domains, several collaborative actions are defined: *2load-carton*, *2unload-carton*, *2load-furniture* and *2unload-furniture*. Each of these collaborative actions is a composition of exactly two corresponding single-agent primitive actions. Single agent actions that have a collaborative version do not succeed in having their intended effects, but make the agent tired. Only *move-agent*, *pack-appliance*, and *unpack-appliance* can be performed by a tired agent. She can recharge by performing the *consume* action. Collaborative actions have their intended effect and do not make the agent tired.

In the privacy preserving planning settings, the location of an agent and whether she is tired are private to the agent.

5.7.2 Results

Our algorithms were implemented in C++ and built on the publicly available code of Furelos-Blanco, Frances, and Jonsson (2019), except for Rep 3. Experiments were carried out on an Intel Core i5 3.20 GHz with 64-bit processor and 4GB of RAM. A time limit of 30 minutes was set per problem. Our translation schemes generate standard single-agent planning problems if there is no privacy involved in the domain description which is identical to that of CJR's method's output when there are no pre/eff interactions and privacy, except for a small overhead associated with maintaining data-structures required to detect and correctly handle inputs with *Pre/Eff* Interactions. Their implementation does not address *Pre/Eff* interactions, and can return incorrect solutions or miss valid solutions when such interactions exist.

For each problem, the translation step generates a single-agent problem, which is given to Fast-Downward (FD) [37]. The search approach used in FD is *lazy-greedy* with h_{FF} heuris-

Ins (#agents)	Size	CJR's			Rep 2: Well-Formed		
		Length	Makespan	Time (s)	Length	Makespan	Time (s)
P01 (5)	19		**U**		8	4	0.075
P02 (4)	52	34	7	31.1	43	36	35.4
P03 (10)	198	58	58	11.6	58	58	12.1
P04 (20)	208	135	131	507.9	139	131	381.4
P05 (6)	444		**U**		69	68	18.5
P06 (10)	446	96	96	258.8	96	96	263.4
P07 (10)	734		**TO**			**TO**	

Table 5.2 Sorted by the problem size the table compares CJR's approach and our well-formed approach with object constraints in the Maze domain [23]. U stands for unsolved and TO stands for a timeout.

tic [39]. We note that all tables reflect solution time without compilation time. Compilation time is negligible, taking small fractions of a second. The size of the translated domain is roughly four times larger.

We note that our compilation code does not automatically add mutex constraints between the actions, and these have been added manually. Defining mutex relations between ground actions is not hard, and could, in principle, be added following the generation of ground actions by the FD solver. However, figuring out mutex relations properly in a domain with ungrounded actions is non-trivial, and is currently not a part of our implementation. In the *general* approach (Rep 3), since the planner can apply any action that manipulates any object or object set, there are many mutex actions and propositions. In Rep 1 and Rep 2, since only actions that manipulate the same objects are allowed concurrently, fewer mutex actions and propositions are needed. Moreover, as we noted, additional information regarding consistency must be supplied in some domain. For example a box can be pushed to two different locations by two agents simultaneously. Logically, the two effects *(at Box* loc_1*)* and *(at Box* loc_2*)* are consistent, and their inconsistency must be declared explicitly. This is done by adding explicit mutex relationship between them.

Table 5.1 shows how our compilation algorithm scales in each domain with no privacy. Each major column represents plan length, makespan, and solution time in seconds of one of our variants.

Comparing Rep 1 and Rep 2, we see that the method described in this chapter, scales a bit worse than the version described in [82], when we force the use of single object set in each multi-action. It is most likely because of the stricter requirement of using well-formed multi-action, and agents acting in parallel might lead to a potentially exponential blowup since, in Rep 2, we have many more explicitly defined collaborative actions. However, it is a tradeoff for building well-formed multi-actions, which are much more intuitive than well-defined multi-actions. Somewhat surprising to us, removing the restriction of multi-actions that focus on a single set of objects, leads to much better performance (Rep 3). We initially expected that this would lead to a much larger branching factor, and hence worse performance.

Another observation is that Rep 2 plans are often longer than Rep 1 plans, although their makespan is often shorter. On the one hand, every well-formed plan is also, essentially well-defined, so Rep 2 has more flexibility, and should be able to produce shorter plans. On the other hand the well-formedness constraint forces us to define actions such as *3push* and *4push,* which in Rep 1 would require two actions to represent (e.g., *(2push,push)* and *(2push,2push)*). But these seem to be minor issues. One must recall that we are not using an optimal planner to solve the problems, and that different heuristics have different behavior, and how they interact with a particular domain description is not well understood. When we rerun some of the instances using the LAMA2011 heuristic [73] instead of the FF heuristic, we indeed observed such sensitivity. On some problems in which Rep 1 produces shorter plans than Rep 2 using the FF heuristic, it generated longer plans than Rep 2 when using the LAMA2011 heuristic, and vice versa.

The results in the BoxPushing domain are somewhat different from those of the other domains. There, Rep 1 scales much better than Rep 3. The issue seems to be memory. To verify this, we run Rep 3 using 8GB instead of 4GB, yet even this was not sufficient for the larger instances. FD grounds everything before it actually starts the search process. Due to many universal and existential quantifiers and propositions in the compiled problems, the memory explodes too early in the *general* case. Quantifiers also affect the performance of the planner in the scenario of *well-formed* multi-actions with constraints. However such memory related issues only appeared in the Boxpushing domain; in the other three domains our general approach is much faster than the other two, but plans generated are often a little longer. Note also that when memory was not an issue, Rep 3 was and Rep 1 are quite similar in this domain.

Following Table 5.1, in the Maze domain, we can see that plan length obtained for problems varies from 12 to 125, however, the sizes of the problems do not vary that much. We hypothesize that it is not always correlated with the difficulty of planning (which is the number of objects), perhaps the nature of the objects and how they constrain the possible path plays a more significant role. The nature is also characterized by the degree of interaction between action.

In Table 5.1, for problem P02, with the maximum bound 4 on each boat, the well-formed case (Rep 2) is almost 5 times slower compared to Rep 1, perhaps produced equal length plans. We can also see that, for problems P03, the Rep 2 took more time (almost 3 times) than Rep 1 as there were 5 agents acting, and that would have increased the total number of grounded actions given that there were several higher order collaborative actions in Rep 2. However, for P04 with 4 agents, Rep 1 is slower than Rep 2 by more than three times.

In the Tablemovers domain, the *well-formed* approach is the worst performer among the three, it usually found longer plans and took more time. Also, some problems were unsolvable using this compilation approach, but they were solved by the other two approaches, and P03. P05 which contain 6 agents, were solved only by the general approach.

In the Apartmentmovers domain a similar trend emerges. Some problems solvable by the *well-defined* approach and general *well-formed* were unsolved when only restricted *well-formed* multi-actions were allowed. P06 and P07 differ only in the addition of an extra agent in P07. The well-defined case solved P07 when we relaxed the upper time bound slightly. In some problems we found that adding another agent in the domain makes them unsolvable for the well-formed case, for example, problems P02 and P03. This is because the Rep2 has more collaborative actions in general than the Rep1.

Table 5.2 shows a comparison of the compilation approach by Crosby, Jonsson, and Rovatsos (2014) and our compilation approach that enforces a multi-action to be well-formed with object constraints as described by CJR. These approaches are compared in the Maze domain. We use the exact domain description for this comparison in which there are only single-agent actions. We note that if the effect of a collaborative action is not different than the union on the effects of its individual components then it is not needed to define the collaborative action. This can be captured by object constraints as described in [23]. For example, suppose that the given constraints on a bridge is (1,10), according to our formalism, multi-action $\langle a_{start}$, *cross, cross, cross,* $a_{end} \rangle$ is an example of a well-formed multi-action.

We created two example problems, P01 and P05 in Table 5.2, such that to solve those, agents must use actions that interfere with each other. For example two agents must cross a bridge simultaneously, as the bridge collapses after its first usage. These problems were unsolved by CJR's compilation approach while our approach solves them both since action interactions are explicitly tackled in our compilation approach. In general our approach adds up several predicates to the compiled domain description to support the notion of well-formed multi-actions and action interactions. These new predicates take extra time to ground before the actual search process starts (in FD). For problems P03 and P06, both the approaches come up with the exact same plans, but our approach was a bit slower. For problem P04, there were 20 agents, their approach took more than 500 seconds to solve, while for our approach it took roughly 380 seconds. For this problem, the solution plan obtained for the well-formed case contains a multi-action with four agents simultaneously crossing a bridge.

Domain	Ins (#agent)	Size	Time (sec)		Length	Makespan
			Centralized	Distributed		
Box-Pushing	P_{dp}01 (3)	7	0.7	2.9	8	4
	P_{dp}02 (4)	9	2.4	5.72	12	7
	P_{dp}03 (4)	13	321.0	322.2	26	16
	P_{dp}04 (5)	12	420.9	766.6	25	18
	P_{dp}05 (5)	13	691.3	1497.1	39	25
Maze	P_{dp}01 (3)	49	8.0	62.2	3	1
	P_{dp}02 (4)	50	10.7	270.1	25	22
	P_{dp}03 (4)	51	13.0	104.8	17	10
	P_{dp}04 (5)	49	15.7	407.9	17	10
	P_{dp}05 (5)	52	18.4	142.7	26	15
Table-Movers	P_{dp}01 (3)	9	0.7	2.9	7	5
	P_{dp}02 (4)	11	2.8	23.1	20	16
	P_{dp}03 (4)	13	5.6	65.5	28	22
	P_{dp}04 (5)	16	4.6	-	-	-
	P_{dp}05 (5)	15	39.9	-	-	-
Apartment-movers	P_{dp}01 (3)	14	15.7	12.8	16	10
	P_{dp}02 (4)	24	47.0	241.0	31	23
	P_{dp}03 (4)	32	35.0	301.0	36	28
	P_{dp}04 (5)	34	200.0	254.5	44	38
	P_{dp}05 (5)	36	160.9	781.6	26	20

Table 5.3 GPPP's performance on the compiled domains with privacy. *Size* shows the number of objects appeared in each problem including the agents. The column *centralized* shows the time taken to find a centralized solution with no privacy, is compared against the distributed case.

This multi-action (a.k.a. joint action for CJR's approach) is not possible to obtain in a plan by CJR's compilation approach. The planner consequently found relatively smaller plan and quicker as well. It timed out for P07 for both the approaches.

Table 5.3 shows results for distributed PPP, for which we used the distributed PPP solver GPPP [53] with the distributed h_{FF} heuristic. Note that currently GPPP has difficulty in supporting the quantifiers present in compiled domain descriptions and several other syntax related issues. In this table we present results for *well-defined* case. Once a distributed privacy preserving problem is generated, we need to remove some of quantified propositions and add actions to make the compiled problem compatible for GPPP. The idea here is to show that our compilation approaches are supported by a PPP planner. Although a lot of manual work is required to do this currently. We do not present results for PPP case for other two approaches. However, one can relate a possible outcomes/results for those two approaches, with the results obtained in the *centralized* (Table 5.1) case in all three cases.

GPPP is far less optimized than FD (on single-agent problems it was 123 times slower, with average ratio per domains ranging from 40 to 287). Hence, we used simpler problems than in Table 5.1. For each problem, beyond showing the results of running GPPP on the compiled problem, we also describe (*centralized*) the running time when the problem is solved by GPPP with a single-agent (that has access to all actions). This gives a sense of the relative difficulty associated with privacy, which we can see is non-negligible, generally between 4-20 times slower. The gap in *Table-Mover* is largest, and *Box-Pushing* and *Apartment-Mover*, smallest.

5.8 Discussion

The problem of modeling joint actions raises various semantic issues, some of which we tried to tackle in this chapter. But subtle issues remain. Consider for example our *2push* action. We defined it as the result of two agents pushing the same box. Thus, if the *push* action is implemented using some code in a robot, we're saying that *2push* is the result of both robots activating this code concurrently on the same box. But is *2push* really two single *push* actions? When two agents push a box together, they need to coordinate with each other. Indeed, a more refined model of this situation might call for each agent executing a different action than a simple *push*.

There are a number of ways of addressing this issue. A simple solution is to ignore it, and assume that at our current abstraction level, these are indeed the same actions. Another solution is to add a new single agent action *joint-push* with a cardinality constraint that requires at least two agents to carry it out concurrently. Notice that this is a different type of constraint from the standard CJR cardinality constraint – the requirement is not that at least two agents manipulate an object, but that at least two agent execute this action concurrently.

With this solution in mind, *2push* would now consist of two *joint-push* actions. This idea can be taken farther: we can also have coordinated actions dealing with multiple objects. For example, two agents, each pushing a box, but where the boxes are adjacent and the result is somehow coordinated. This would actually require a new version of *push* with two box arguments, where each agent actually changes the location of one box only.

A potential practical way of addressing this without having to specify so many single-agent actions is to allow for primitive collaborative actions – i.e., one for which no single agent elements are actually specified. But essentially, for this to work out semantically, it seems that each such collaborative action implicitly defines special single-agent actions.

Another practical issue is the need to address multiple agents doing the same action, as in *push*, *2push*, *3push*, etc. Observe that, in fact, we cannot combine two *2push* actions in a multi-action because it would not be well-formed. For example, $\{\mathit{2push}(a_1,a_2,b), \mathit{2push}(a_3,a_4,b)\}$ has $\mathit{push}(a_1,b), \mathit{push}(a_3,b)$ among its primitive elements, which together form the collaborative action $\mathit{2push}(a_1,a_3,b)$. Thus, in the worst case, we need a special k*push* action for every $k \leq n$. If from some point on the addition of agents does not impact the effects on individual agents, we could capture this using a single-action *multi-push* with the same type of constraint as above, where at least k agents are required to execute *multi-push*. We note that such constraints are naturally expressed in the framework of Boutilier and Brafman (1997). However, as noted, this approach appears challenging for forward-search planning given that the effect of an action is unknown unless the other actions are specified. Thus, it would be difficult to get good heuristics for methods that attempt to add single-agent actions incrementally.

The above problem impacts the succinctness of the language. The concept of well-formed actions forces the modeler to explicitly specify the effect of combined actions. In many cases, this is justified. But there may be cases where this is redundant. For example, suppose that there are two collaborative action (a_1,a_2) and (a_2,a_3). In this case, $((a_1,a_2),a_3)$ is not well-formed, and to execute these three actions, we must have an explicit collaborative action that contains all three single agent actions. But if the effect on the agent performing a_2 in (a_1,a_2) and in (a_2,a_3) is identical, and if the effect on the agent performing a_3 in (a_2,a_3) is identical to that of just a_3, then the combination $((a_1,a_2),a_3)$ makes some sense. This, however, appears a somewhat esoteric case, which can be supported by preprocessing – automatically suggesting to the user such combinations and forming appropriate collaborative actions for them. Supporting this in the compilation process directly seems difficult.

The above issues might lead future work to reconsider our definition of well-formed multi-action, allowing for more flexibility. We again note that originally, we used a weaker definition of *well-defined* actions [82]. Formally, if $a_c = \{a_1,\ldots,a_l\}$ and $a'_c = \{a'_1,\ldots,a'_m\}$ are two multi-actions, we say that a_c is *subsumed* by a'_c if: (1) $e(a'_c) = e(a_c)$; (2) for every $a_i \in a_c$ there is some $a'_j \in a'_c$ such that $e(a_i) \subseteq e(a'_j)$; and (3) $m < l$. That is, both multi-actions involve the same set of elements, and moreover, for every member of a_c, there is a member of a'_c that contains all the elements of a_c, and that this containment is strict in at least one case (and hence, $m < l$). A multi-action is *well-defined* if it is not subsumed by any other multi-action. To ensure the generation of well-defined multi-actions, we used a general compilation scheme exactly similar to the scheme shown in Section 5.5.3, except for just one major change. In Step $2(d)$, if $e(a_c) = \{a,a_1,\ldots,a_k\}$ for some collaborative action a_c, we added only $\neg(p_{a_1} \wedge \cdots \wedge p_{a_k})$ as a precondition to a. Perhaps it is not hard to see that every well-formed action is well-defined, but not vice versa. In fact, if we allow well-defined multi-actions, multiple multi-actions can have the same set of primitive elements. Thus, Lemma 1 would no longer be true. For example, assuming *push* and *2push*, $\{2push(a_1,a_2,b)$, $2push(a_3,a_4,b)\}$, $\{2push(a_1,a_3,b)$, $2push(a_2,a_4,b)\}$, $\{2push(a_1,a_4,b)$, $2push(a_3,a_4,b)\}$ are well-defined and contain the same primitive elements. None of them is well-formed, though. In this symmetric case, it may seem unharmful, but consider: $\{push(a_1,b)$, $2push(a_2,a_3,b)\}$, $\{push(a_2,b)$, $2push(a_1,a_3,b)\}$, $\{push(a_3,b)$, $2push(a_1,a_2,b)\}$. All have the same primitive elements, all correspond to the same joint-action, and if *push* and *2push* have different effects (e.g., following *push* the agent is tired), we have an ambiguous semantic for a joint action. For this reason, we feel the new definition is cleaner semantically.

A possible direction for future work is to use our language and semantic to support planning using input in BB's format. The idea is the following: given a specification using BB's method, we define a set of collaborative actions such that the semantics of the resulting domain captures BB's semantics for the original domain. The benefit of this is that the effect of collaborative actions is independent of other actions, and so they can be introduced into the plan incrementally without altering their effects, which seems more suitable for existing heuristics. There seem to be two issues to tackle here. First, one must deal with negative concurrency constraints – an effect of action a may take place only if a' is not performed concurrently. We believe this can be enforced using suitable preconditions, in the spirit of Step $2(a)$ and Step $2(b)$ in our encoding. Second, one would have to show that the well-formedness requirement does not cause us to ignore some combinations that BB's semantics would allow.

5.9 Summary

In this section we will summarize this chapter. We presented a new approach, which is built upon earlier work, to modeling and planning with interacting actions and privacy that is intuitive and supports efficient planning. A key property of our semantics is monotonicity in the effects of joint actions – an element added to a multi-action does not modify the effects of actions added earlier. We described a compilation scheme from our input language to single-agent planning and distributed privacy-preserving planning, which is the first to extend both the language and algorithms for classical planning to handle these issues.

```
( define  (domain boxpushing)
(: requirements  : typing )
(: types  agent  − object  bridge  − object  box − object  location  − object )
(:  predicates
     (box−at ?b  − box ?loc  −  location )
     (pos−box−at ?b  − box ?loc  −  location )
     (neg−box−at ?b  − box ?loc  −  location )
     ( intact  ?box − box)
     (pos− intact  ?box − box)
     (neg− intact  ?box − box)
     ( at  ?agt  − agent  ?loc  −  location )
     (pos−at  ?agt  − agent  ?loc  −  location )
     (neg−at  ?agt  − agent  ?loc  −  location )
     (has−bridge  ?br  − bridge  ?loc1  −  location  ?loc2  −  location )
     (pos−has−bridge  ?br  − bridge  ?loc1  −  location  ?loc2  −  location )
     (neg−has−bridge  ?br  − bridge  ?loc1  −  location  ?loc2  −  location )
     ( tired  ?agt  − agent)
     (pos− tired  ?agt  − agent)
     (neg− tired  ?agt  − agent)
     (taken  ?agt  − agent)
     ( in )
     (p−cross  ?a1  − agent  ?br  − bridge  ?loc1  −  location  ?loc2  −  location )
     (p−push ?a1  − agent  ?b  − box ?loc1  −  location  ?loc2  −  location )
     (p−2push ?a1 ?a2  − agent  ?b  − box ?loc1  −  location  ?loc2  −  location ))
(: action  a− start
     : parameters ()
     : precondition (and  (not  ( in )))
     : effect (and  ( in )))
(: action  cross
     : parameters  (?a1  − agent  ?b  − bridge  ?loc1  −  location  ?loc2  −  location )
     : precondition  (and  ( in )( at  ?a1  ?loc1)(has−bridge  ?b  ?loc1  ?loc2)(not  (taken  ?a1)))
     : effect  (and  (pos−at  ?a1  ?loc2)(neg−has−bridge  ?b  ?loc1  ?loc2)
                  (neg−has−bridge  ?b  ?loc2  ?loc1)(neg−at  ?a1  ?loc1)
                  (p−cross  ?a1  ?b  ?loc1  ?loc2)(taken  ?a1)))
(: action  push
     : parameters  (?a1  − agent  ?b  − box ?loc1  −  location  ?loc2  −  location )
     : precondition  (and  ( in )( at  ?a1  ?loc1)(not  ( tired  ?a1))(box−at  ?b  ?loc1)(not  (taken  ?a1))
                          ( forall   (?a3  − agent  ?a4  − agent)(not  (p−2push ?a3 ?a4  ?b  ?loc1  ?loc2)))
                          ( forall   (?a3  − agent)(not  (p−push ?a3  ?b  ?loc1  ?loc2))))
     : effect  (and  (pos− tired  ?a1)(neg− intact  ?b)(p−push ?a1  ?b  ?loc1  ?loc2)(taken  ?a1)))
(: action  2push
     : parameters  (?a1  − agent  ?a2  − agent  ?b  − box ?loc1  −  location  ?loc2  −  location )
     : precondition  (and  ( in )(not  (=  ?a1  ?a2))( at  ?a1  ?loc1)( at  ?a2  ?loc1)(box−at  ?b  ?loc1)
                          (not  ( tired  ?a1))(not  ( tired  ?a2))(not  (taken  ?a1))(not  (taken  ?a2))
                       ( forall   (?a3  − agent)(not  (p−push ?a3  ?b  ?loc1  ?loc2))))
     : effect  (and  (pos−at  ?a1  ?loc2)(pos−at  ?a2  ?loc2)(neg−at  ?a1  ?loc1)(neg−at  ?a2  ?loc1)
                  (pos−box−at  ?b  ?loc2)(pos− intact  ?b)(neg−box−at  ?b  ?loc1)
                  (p−push ?a1  ?b  ?loc1  ?loc2)(p−2push ?a1  ?a2  ?b  ?loc1  ?loc2)(taken  ?a1)(taken  ?a2)))
(: action  a−end  ....)  )
```

Listing 5.3 The Compiled Domain

Chapter 6

Conclusions and Future Challenges

This chapter summarizes the thesis by highlighting our research contributions, and then describes some challenges for future work.

6.1 Summary

This work explores the problem of collaborative MAP with partial observability, interacting actions, and privacy. We used the Qualitative Dec-POMDP model [18] to capture the MAP problem with partial observability and under uncertainty, a model introduced as an alternative to Dec-POMDPs [7, 64, 79], that replaces the probability distributions over possible states with qualitative sets of states [18]. Originally, QDec-POMDPs were shown to scale better than Dec-POMDPs, but not substantially so. In this thesis, we proposed two new algorithms that share the same high-level factored framework inspired by the factored approaches in classical planning to enhance the scalability of QDec-POMDPs compared with contemporary (*Qualitative*) Dec-POMDPs; these algorithms demonstrate far greater scalability.

The first approach, QDec-FP, starts by treating the entire MA system as a single-agent system, called the team-problem. A solution to the team-problem generated by a contingent solver is called the team-solution. We then project the team-solution to the appropriate part of each agent. Then, each agent tries to *fix* their projected tree so that the agent can execute it online. Fixing their projected tree requires inserting additional actions and replacing the other agents' sensing actions with their own sensing actions. If all agents succeed, the approach aligns the solution of each projected problem.

The QDec-FP's team planning process relaxes the need to maintain different information states for different agents. Instead, the team planning only manages a single belief state for multiple agents. Therefore, while fixing their projected subtrees, agents might be required to act under the conditions they cannot know, which has often led to *failure* in planning at the agent-level. To alleviate this issue, we proposed to model "agent-specific knowledge" during team planning. This new idea not only helped to generate informed team-solutions, but also helped model communication between agents. We model *signaling* – an implicit form of communication in which agents share knowledge by changing the state of the world and use that in team planning, which cannot be modeled in practice in QDec-FP.

In Chapter 3, we showed that QDec-FP scaled much better than the IMAP approach, the prior state-of-the-art algorithm. In Chapter 4, we showed that the QDec-FPS solver, which models individual agents' knowledge, enhanced the scalability of the QDec-POMDP framework even further. It solved problems that needed *signaling* to be solved, which were practically not solvable by QDec-FP.

Therefore, one can say that seeking a good factored algorithm sometimes helps to better understand the underlying *abstraction* used for coordinating the agents and its shortcomings. This understanding may lead to insights for creating better abstractions, e.g., an approach with a better pruning technique, heuristics, etc.

As we sought to improve the QDec-FP approach, we realized that team planning generates an abstraction that is too strong. The obtained abstract solution allows all agents to execute their public actions based on the results obtained by a "private" action by a single agent. In our case, that private action is that agent's own sensing action. This often leads to failure at the agent-level planning. A better abstraction is achieved when we use agent-specific knowledge-modeling during team planning in QDec-FPS, that is, by allowing the team planner (unlike in the previous case) to apply only those actions related to the agent, who applied its sensing action earlier in the plan tree. However, this is not true when an agent applies a regular public action, say a push, and in that case, the effects/results are shared with all agents at the level of team planning. In the next section, one of the listed future works follows the same trace of thoughts.

The latter part of the thesis explored a formalism for specifying joint-actions in a compositional way, which is intuitive, and a compilation-based approach to planning with interacting actions, as well as privacy. It also highlights and discusses subtle issues that arise when attempting to model and plan with interacting actions.

6.2 Future Work

Many challenges remain for future research, which we briefly present in this section. We describe some ideas that can be considered as immediate future extensions of our current approaches, while others are for the long run.

6.2.1 MAP with Partial Observability and under Uncertainty

There are many important questions that remain unanswered in the *Qualitative* Dec-POMDPs formalism. We hope that future research will continue investigating this topic. An immediate extension of our factored approach is a more efficient backtracking mechanism. A sound approach that augments the MAP domain with learned *no-goods* could help generate team plans that are likely to succeed even more. Together they are expected to enhance the scalability of QDec-POMDP even further.

One could also restrict an agent from applying specific actions (ideally "public") or decide that the agent must apply specific actions (in a certain order) under certain conditions during team planning, which would help generate an even more informed abstract plan (a team solution). However, we note that to achieve this, one needs to change the underlying planner, which is CPOR in the case of QDec-FP/S, in a certain way. One can expect that a sound mechanism would enable the factored approach to go for an "early" backtrack. However, one would need to devise a sound approach to learn *no-goods* for domain augmentation in this case, too.

Even with improved knowledge modeling, the underlying planner (CPOR) can generate a team plan that requires an agent to act differently for some value of a variable that it cannot observe. To be precise, these are the cases where one agent does not need the exact value of this variable to execute its actions in the team plan (*i.e.*, that exact value is not required as a precondition of its actions). Such a team plan cannot be decomposed and solved by this agent. The situation arises here because a team plan is represented as a tree (or a graph), and is not linear, so the decision-making and reasoning required under partial observability and uncertainty is much more involved.

Modeling some form of communication ("signaling") can handle such scenarios under specific circumstances where an agent can share some information that another agent cannot learn on its own. Let us consider an example: Suppose that the agent φ_1 wakes-up at 7AM, then another agent φ_2 would pick φ_1 up from *station-A*, at some time point, but if φ_1 wakes-up later, φ_2 picks φ_1 up from *station-B*, at some other time point. To *pick-up* φ_1 from a station, it must be there at the station, which also means that φ_1 is initially not present at either station. Moreover, φ_2 cannot observe the wake-up time of φ_1, but it can sense φ_1's presence at a station. Here the two decision points, (a) when φ_1 wakes up, and (b) at which station φ_1 would be present after some time points to get picked up, are intertwined. Our current approaches cannot generate an abstract team plan that supports proper coordination of the actions of the agents. Nevertheless, once agents fail to convert this team plan into a solution plan, one can post-process the team plan to learn some *relations* between the *two decision points* and use them in the next iterations.

Another possible direction is to devise a new representation for joint-plans that will replace the current plan-tree used by us (and by most other approaches) with its fully-ordered branches. An in-depth study of the joint-policy trees obtained while solving specific benchmark domains motivates us to seek a different representation to that which we currently use. Moreover, as the decision-making and reasoning required under partial observability and uncertainty are much more involved, we also think that a better representation would help minimize the number of backtracks currently needed to solve the MAP problem. That would enhance the overall scalability and applicability of QDec-POMDPs.

A big challenge awaiting is a thorough study of the applicability of the factored planning approach in non-deterministic domains. We note that our proposed methods have not been tested on non-deterministic domains.

6.2.2 MAP with Interacting Actions

For future research in this direction, one could consider the input in the format suggested by Boutilier and Brafman (1997) in [12], and use our language and semantics to support multi-agent planning. The idea is the following: given a specification using Boutilier and Brafman's method, we define a set of collaborative actions such that the semantics of the resulting domain capture Boutilier and Brafman's semantics for the original domain. The advantage of this is that the effect of collaborative actions is independent of other actions, and therefore they can be introduced into the plan incrementally without altering their effects, which seems more suitable for existing classical heuristics.

References

[1] Albore, A., Palacios, H., and Geffner, H. (2009). A translation-based approach to contingent planning. In Boutilier, C., editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1623–1628.

[2] Amir, E. and Engelhardt, B. (2003). Factored planning. In Gottlob, G. and Walsh, T., editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 929–935. Morgan Kaufmann.

[3] Anderson, C. R., Smith, D. E., and Weld, D. S. (1998). Conditional effects in graphplan. In Simmons, R. G., Veloso, M. M., and Smith, S. F., editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, Pennsylvania, USA, 1998*, pages 44–53. AAAI.

[4] Baral, C. and Gelfond, M. (1997). Reasoning about effects of concurrent actions. *J. Log. Program.*, 31(1-3):85–117.

[5] Bazinin, S. and Shani, G. (2018). Iterative planning for deterministic QDec-POMDPs. In *GCAI 2018*, pages 15–28.

[6] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37.

[7] Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27:819–840.

[8] Bernstein, D. S., Zilberstein, S., and Immerman, N. (2000). The complexity of decentralized control of markov decision processes. In Boutilier, C. and Goldszmidt, M., editors, *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000*, pages 32–37. Morgan Kaufmann.

[9] Blum, A. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

[10] Bonet, B. and Geffner, H. (2011). Planning under partial observability by classical replanning: Theory and experiments. In Walsh, T., editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1936–1941. IJCAI/AAAI.

[11] Bonet, B. and Geffner, H. (2014). Belief tracking for planning with sensing: Width, complexity and approximations. *J. Artif. Intell. Res.*, 50:923–970.

[12] Boutilier, C. and Brafman, R. I. (1997). Planning with concurrent interacting actions. In *Proc. of the 14th National Conference on AI (AAAI '97)*, pages 720–726.

[13] Brafman, R. I. (2015). A privacy preserving algorithm for multi-agent planning and search. In Yang, Q. and Wooldridge, M. J., editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1530–1536. AAAI Press.

[14] Brafman, R. I. and Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, pages 28–35.

[15] Brafman, R. I. and Domshlak, C. (2013). On the complexity of planning for agent teams and its implications for single agent planning. *Artif. Intell.*, 198:52–71.

[16] Brafman, R. I. and Shani, G. (2012). Replanning in domains with partial information and sensing actions. *J. Artif. Intell. Res.*, 45:565–600.

[17] Brafman, R. I. and Shani, G. (2016). Online belief tracking using regression for contingent planning. *Artif. Intell.*, 241:131–152.

[18] Brafman, R. I., Shani, G., and Zilberstein, S. (2013). Qualitative planning under partial observability in multi-agent domains. In *AAAI'13*.

[19] Brafman, R. I. and Zoran, U. (2014). Distributed heuristic forward search with interacting actions. In *Proc. of the 2nd ICAPS Workshop on Distributed and Multi-Agent Planning*.

[20] Bryce, D., Kambhampati, S., and Smith, D. E. (2006). Planning graph heuristics for belief space search. *J. Artif. Intell. Res.*, 26:35–99.

[21] Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204.

[22] Cassandra, A. R. (1998). Exact and approximate algorithms for partially observable markov decision processes.

[23] Crosby, M., Jonsson, A., and Rovatsos, M. (2014). A single-agent approach to multiagent planning. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 237–242.

[24] Crosby, M. and Petrick, R. P. (2014). Temporal multiagent planning with concurrent action constraints. In *workshop on Distributed and Multi-Agent Planning (online proceedings)*. ICAPS 2014.

[25] Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, pages 169–181.

[26] Durfee, E. H. (2001). Distributed problem solving and planning. In Luck, M., Marík, V., Stepánková, O., and Trappl, R., editors, *Multi-Agent Systems and Applications, 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School, EASSS 2001, Prague, Czech Republic, July 2-13, 2001, Selected Tutorial Papers*, volume 2086 of *Lecture Notes in Computer Science*, pages 118–149. Springer.

[27] Durfee, E. H., Lesser, V. R., and Corkill, D. D. (1987). Coherent cooperation among communicating problem solvers. *IEEE Trans. Computers*, 36(11):1275–1291.

[28] Edelkamp, S. (2002). Symbolic pattern databases in heuristic search planning. In Ghallab, M., Hertzberg, J., and Traverso, P., editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France*, pages 274–283. AAAI.

[29] Ephrati, E. and Rosenschein, J. S. (1997). A heuristic technique for multi-agent planning. *Ann. Math. Artif. Intell.*, 20(1-4):13–67.

[30] Fabre, E. and Jezequel, L. (2009). Distributed optimal planning: an approach by weighted automata calculus. In *Proceedings of the 48th IEEE Conference on Decision and Control, CDC 2009, combined withe the 28th Chinese Control Conference, December 16-18, 2009, Shanghai, China*, pages 211–216. IEEE.

[31] Fabre, E., Jezequel, L., Haslum, P., and Thiébaux, S. (2010). Cost-optimal factored planning: Promises and pitfalls. In Brafman, R. I., Geffner, H., Hoffmann, J., and Kautz, H. A., editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 65–72. AAAI.

[32] Fikes, R. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. In Cooper, D. C., editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 608–620. William Kaufmann.

[33] Fox, M. and Thiébaux, S. (2009). Advances in automated plan generation. *Artif. Intell.*, 173(5-6):501–502.

[34] Furelos-Blanco, D. and Jonsson, A. (2019). Solving multiagent planning problems with concurrent conditional effects. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7594–7601. AAAI Press.

[35] Ghallab, M., Nau, D. S., and Traverso, P. (2004). *Automated planning - theory and practice*. Elsevier.

[36] Grosz, B. J. and Sidner, C. L. (1986). Attention, intentions, and the structure of discourse. *Comput. Linguistics*, 12(3):175–204.

[37] Helmert, M. (2006). The fast downward planning system. *J. Artif. Int. Res.*, 26(1):191–246.

[38] Ho, Y.-C. (1980). Team decision theory and information structures. *Proceedings of the IEEE*, 68(6):644–654.

[39] Hoffmann, J. and Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.*, 14(1):253–302.

[40] Jennings, N. R., Sycara, K. P., and Wooldridge, M. J. (1998). A roadmap of agent research and development. *Auton. Agents Multi Agent Syst.*, 1(1):7–38.

[41] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134.

[42] Kolobov, A., Mausam, and Weld, D. S. (2010). Classical planning in MDP heuristics: with a little help from generalization. In Brafman, R. I., Geffner, H., Hoffmann, J., and Kautz, H. A., editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 97–104. AAAI.

[43] Komarnitsky, R. and Shani, G. (2016). Computing contingent plans using online replanning. In *AAAI'13*, pages 3159–3165.

[44] Komenda, A., Stolba, M., and Kovacs, D. L. (2016). The international competition of distributed and multiagent planners (codmap). *AI Mag.*, 37(3):109–115.

[45] Kovacs, D. L. (2012). A multi-agent extension of pddl3.1. In *Proc. of of the 3rd Workshop on the International Planning Competition (IPC 2012).*

[46] Kumar, A., Zilberstein, S., and Toussaint, M. (2011). Scalable multiagent planning using probabilistic inference. In *IJCAI 2011*, pages 2140–2146.

[47] Lansky, A. L. and Getoor, L. (1995). Scope and abstraction: Two criteria for localized planning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1612–1619. Morgan Kaufmann.

[48] Lin, F. and Shoham, Y. (1992). Concurrent actions in the situation calculus. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 590–595.

[49] Littman, M. L., Goldsmith, J., and Mundhenk, M. (1998). The computational complexity of probabilistic planning. *Journal of AI Research*, 9:1–36.

[50] Maliah, S., Brafman, R. I., Karpas, E., and Shani, G. (2014a). Partially observable online contingent planning using landmark heuristics. In Chien, S. A., Do, M. B., Fern, A., and Ruml, W., editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI.

[51] Maliah, S., Shani, G., and Brafman, R. I. (2016). Online macro generation for privacy preserving planning. In Coles, A. J., Coles, A., Edelkamp, S., Magazzeni, D., and Sanner, S., editors, *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, pages 216–220. AAAI Press.

[52] Maliah, S., Shani, G., and Stern, R. (2014b). Privacy preserving landmark detection. In Schaub, T., Friedrich, G., and O'Sullivan, B., editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 597–602. IOS Press.

[53] Maliah, S., Shani, G., and Stern, R. (2017). Collaborative privacy preserving multi-agent planning - planners and heuristics. *Autonomous Agents and Multi-Agent Systems*, 31(3):493–530.

[54] Monderer, D. and Shapley, L. S. (1996). Potential games. *Games and economic behavior*, 14(1):124–143.

[55] Muise, C. J., Belle, V., and McIlraith, S. A. (2014). Computing contingent plans via fully observable non-deterministic planning. In Brodley, C. E. and Stone, P., editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2322–2329. AAAI Press.

[56] Nair, R., Tambe, M., Yokoo, M., Pynadath, D. V., and Marsella, S. (2003). Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In Gottlob, G. and Walsh, T., editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 705–711. Morgan Kaufmann.

[57] Nissim, R. and Brafman, R. I. (2012). Multi-agent a* for parallel and distributed systems. In van der Hoek, W., Padgham, L., Conitzer, V., and Winikoff, M., editors, *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, pages 1265–1266. IFAAMAS.

[58] Nissim, R. and Brafman, R. I. (2014). Distributed heuristic forward search for multi-agent planning. *J. Artif. Intell. Res.*, 51:293–332.

[59] Nissim, R., Brafman, R. I., and Domshlak, C. (2010). A general, fully distributed multi-agent planning algorithm. In van der Hoek, W., Kaminka, G. A., Lespérance, Y., Luck, M., and Sen, S., editors, *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3*, pages 1323–1330. IFAAMAS.

[60] Oliehoek, F. A. and Amato, C. (2016). *A Concise Introduction to Decentralized POMDPs*. Springer Briefs in Intelligent Systems. Springer 2016.

[61] Oliehoek, F. A., Spaan, M. T. J., and Vlassis, N. A. (2008a). Optimal and approximate Q-value functions for decentralized POMDPs. *J. Artif. Intell. Res.*, 32:289–353.

[62] Oliehoek, F. A., Spaan, M. T. J., Whiteson, S., and Vlassis, N. A. (2008b). Exploiting locality of interaction in factored Dec-POMDPs. In *AAMAS 2008*, pages 517–524.

[63] Oliehoek, F. A., Witwicki, S., and Kaelbling, L. P. (2021). A sufficient statistic for influence in structured multiagent environments. *J. Artif. Intell. Res.*, 70.

[64] Oliehoek, F. A., Witwicki, S. J., and Kaelbling, L. P. (2012). Influence-based abstraction for multiagent systems. In *AAAI*.

[65] Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *J. Artif. Intell. Res.*, 35:623–675.

[66] Peot, M. A. and Smith, D. E. (1992). Conditional nonlinear planning. In *Artificial Intelligence Planning Systems*, pages 189–197. Elsevier.

[67] Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56.

[68] Pryor, L. and Collins, G. (1996). Planning for contingencies: A decision-based approach. *J. Artif. Intell. Res.*, 4:287–339.

[69] Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley.

[70] Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[71] Pynadath, D. V. and Tambe, M. (2002). Multiagent teamwork: analyzing the optimality and complexity of key theories and models. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 873–880. ACM.

[72] Reiter, R. (1991). The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Lifshitz, V., editor, *AI and Mathematical Theory of Computation: papers in honour of John McCarthy*, pages 359–380.

[73] Richter, S., Westphal, M., and Helmert, M. (2011). Lama 2008 and 2011. In *IPC 2011 Deterministic Track*, pages 117–124.

[74] Rintanen, J. (2004). Complexity of planning with partial observability. In Zilberstein, S., Koehler, J., and Koenig, S., editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 345–354. AAAI.

[75] Rintanen, J., Heljanko, K., and Niemelä, I. (2006). Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080.

[76] Rosenthal, R. W. (1973). A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2(1):65–67.

[77] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education.

[78] Sapena, O., Onaindia, E., and Torreño, A. (2013). Forward-chaining planning with a flexible least-commitment strategy. In Gibert, K., Botti, V. J., and Bolaño, R. R., editors, *Artificial Intelligence Research and Development - Proceedings of the 16th International Conference of the Catalan Association for Artificial Intelligence, Vic, Catalonia, Spain, October 23-25, 2013*, volume 256 of *Frontiers in Artificial Intelligence and Applications*, pages 41–50. IOS Press.

[79] Seuken, S. and Zilberstein, S. (2008). Formal models and algorithms for decentralized decision making under uncertainty. *Auton. Agents Multi Agent Syst.*, 17(2):190–250.

[80] Shani, G. (2018). Advances and challenges in privacy preserving planning. In Lang, J., editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5719–5723. ijcai.org.

[81] Shani, G. and Brafman, R. I. (2011). Replanning in domains with partial information and sensing actions. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 2021–2026.

[82] Shekhar, S. and Brafman, R. I. (2018). Representing and planning with interacting actions and privacy. In de Weerdt, M., Koenig, S., Röger, G., and Spaan, M. T. J., editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 232–240. AAAI Press.

[83] Shekhar, S. and Brafman, R. I. (2020). Representing and planning with interacting actions and privacy. *Artif. Intell.*, 278.

[84] Shekhar, S., Brafman, R. I., and Shani, G. (2019). A factored approach to deterministic contingent multi-agent planning. In Benton, J., Lipovetzky, N., Onaindia, E., Smith, D. E., and Srivastava, S., editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 419–427. AAAI Press.

[85] Shekhar, S., Brafman, R. I., and Shani, G. (2020). Signaling in contingent multi-agent planning. In *Workshop on Epistemic Planning (EpiP) 2020, (online proceedings)*. ICAPS 2020.

[86] Shekhar, S., Brafman, R. I., and Shani, G. (2021). Improved knowledge modeling and its use for signaling in multi-agent planning with partial observability. In *35th AAAI Conference on Artificial Intelligence, AAAI 2021*. AAAI Press (*to appear*).

[87] Shoham, Y. and Leyton-Brown, K. (2009). *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

[88] Singh, M. P. (1994). *Multiagent Systems - A Theoretical Framework for Intentions, Know-How, and Communications*, volume 799 of *Lecture Notes in Computer Science*. Springer.

[89] Spaan, M. T. J. and Melo, F. S. (2008). Interaction-driven markov games for decentralized multiagent planning under uncertainty. In Padgham, L., Parkes, D. C., Müller, J. P., and Parsons, S., editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 1*, pages 525–532. IFAAMAS.

[90] Stolba, M. (2017). *Reveal or Hide: Information Sharing in Multi-Agent Planning*. PhD thesis, Czech Technical University in Prague, Prague, Czech Republic.

[91] Stolba, M. and Komenda, A. (2017). The MADLA planner: Multi-agent planning by combination of distributed and local heuristic search. *Artif. Intell.*, 252:175–210.

[92] Stone, P. and Veloso, M. M. (2000). Multiagent systems: A survey from a machine learning perspective. *Auton. Robots*, 8(3):345–383.

[93] Sycara, K. P. (1998). Multiagent systems. *AI Mag.*, 19(2):79–92.

[94] Sycara, K. P., Pannu, A., Williamson, M., Zeng, D., and Decker, K. (1996). Distributed intelligent agents. *IEEE Expert*, 11(6):36–46.

[95] Tambe, M. (1997). Towards flexible teamwork. *J. Artif. Intell. Res.*, 7:83–124.

[96] Torreño, A., Onaindia, E., Komenda, A., and Stolba, M. (2018). Cooperative multi-agent planning: A survey. *ACM Comput. Surv.*, 50(6):84:1–84:32.

[97] Torreño, A., Onaindia, E., and Sapena, O. (2014). FMAP: distributed cooperative multi-agent planning. *Appl. Intell.*, 41(2):606–626.

[98] Tozicka, J., Jakubuv, J., and Komenda, A. (2014). Generating multi-agent plans by distributed intersection of finite state machines. In Schaub, T., Friedrich, G., and O'Sullivan, B., editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 1111–1112. IOS Press.

[99] Varakantham, P., Kwak, J., Taylor, M. E., Marecki, J., Scerri, P., and Tambe, M. (2009). Exploiting coordination locales in distributed pomdps via social model shaping. In Gerevini, A., Howe, A. E., Cesta, A., and Refanidis, I., editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

[100] Vlassis, N. (2007). Distributed decision making for robot teams. In Badica, C. and Paprzycki, M., editors, *Advances in Intelligent and Distributed Computing, Proceedings of the 1st International Symposium on Intelligent and Distributed Computing, IDC 2007, Craiova, Romania, October 2007*, volume 78 of *Studies in Computational Intelligence*, pages 35–40. Springer.

[101] von Neumann, J. and Morgenstern, O. (1994). *Theory of Games and Economic Behavior (60th-Anniversary Edition)*. Princeton University Press.

[102] Wilkins, D. E. and Myers, K. L. (1998). A multiagent planning architecture. In Simmons, R. G., Veloso, M. M., and Smith, S. F., editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, Pennsylvania, USA, 1998*, pages 154–162. AAAI.

[103] Witwicki, S. J. and Durfee, E. H. (2010). From policies to influences: a framework for nonlocal abstraction in transition-dependent Dec-POMDP agents. In *AAMAS'10*, pages 1397–1398.

[104] Yao, A. C. (1982). Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society.

[105] Yoon, S. W., Fern, A., and Givan, R. (2007). FF-replan: A baseline for probabilistic planning. In Boddy, M. S., Fox, M., and Thiébaux, S., editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, page 352. AAAI.

[106] Yoon, S. W., Fern, A., Givan, R., and Kambhampati, S. (2008). Probabilistic planning via determinization in hindsight. In Fox, D. and Gomes, C. P., editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1010–1016. AAAI Press.

Collaborative Multi-Agent Planning with Partial Observability and Interacting Actions

Shashank Shekhar

The Faculty of Natural Sciences
Department of Computer Science
BEN-GURION UNIVERSITY OF THE NEGEV

Thesis submitted in partial fulfillment of the requirements for the degree of

“DOCTOR OF PHILOSOPHY”

March 2021

תכנון רב-סוכנים שיתופי עם תצפיות חלקיות ופעולות עם תלות הדדית

שאשנק שקאר

הפקולטה למדעי הטבע
המחלקה למדעי המחשב

אוניברסיטת בן–גוריון בנגב

עבודת הגמר הוגשה במילוי חלקי של הדרישות לתואר "דוקטור לפילוסופיה”

מרץ 2021

Research-Student's Affidavit when Submitting the Doctoral Thesis for Judgment

I, Shashank Shekhar, whose signature appears below, hereby declare that:

___ I have written this Thesis by myself, except for the help and guidance offered by my Thesis Advisors.

___ The scientific materials included in this Thesis are products of my own research, culled from the period during which I was a research student.

Date: **14 March 2021** Student's name: **Shashank Shekhar** Signature: Shashank Shekhar

תקציר

תכנון רב-סוכנים שיתופי (MAP) תחת אי-ודאות עם יכולת תצפית חלקית על מצב העולם היא בעיה הידועה כמאוד מורכבת. בעיות MAP כאלה ממודלות לעיתים קרובות כ- Dec-POMDP, או בגרסה נוספת שלה כ- QDec-POMDPs, שהיא למעשה גרסת MAP של תכנון מותנה. מודל QDec-POMDP הוצג בתקווה שהמבנה הלא-הסתברותי היותר פשוט שלו יאפשר פתרון גם עבור בעיות מורכבות יותר. ואכן, לפחות כאשר הפעולות דטרמיניסטיות, אלגוריתם MAP איטרטיבי מאפשר פתרון בעיות מורכבות הרבה יותר מאלגוריתמי Dec-POMDP דומים.

בעבודה זו אנו מתארים שתי גישות חדשות לפתרון בעיות QDec-POMDP דטרמיניסטיות, אשר חולקות רעיון משותף של פירוק הבעיה לחלקים. המוטיבציה לגישות אלו נלקחה מאלגוריתמי תכנון קלאסי הפותרים את הבעיה בחלקים, כאשר הפירוק מתקבל ע"י הגדרת בעיית התכנון כבעיית MAP.

בגישה הראשונה שלנו, אנו מתחילים במציאת פיתרון לבעיית MAP בה תוצאות התצפיות זמינות לכל הסוכנים. זוהי למעשה בעיית תכנון של סוכן יחיד עבור כל הצוות, הנקראת "בעיית צוות". לאחר מכן, אנו מסננים מעץ הפיתרון שהתקבל, עץ-משנה אחד עבור כל סוכן, ונותנים לכל סוכן להפוך את העץ הפרטי שלו לעץ מקומי הניתן לביצוע בזמן אמת. אם כל הסוכנים מצליחים בכך, אנו משלבים את העצים לתכנית משותפת תקפה. אחרת, אנו ממשיכים לחקור את מרחב פתרונות "בעיית הצוות". אנו קוראים לגישה זאת QDec-FP. בגישה השנייה אנו מתארים אלגוריתם המשתמש במידע עשיר יותר על הידע של הסוכנים בכדי לשפר את תהליך תכנון הצוות ובכדי לייצר תוכניות סוכן בודד "מושכלות" יותר עבור כל הסוכנים. מידול הידע של סוכן הבודד תומך גם במודלים של תקשורת בין סוכנים ובתכנון תוך התחשבות בתקשורת זאת . בפרט, אנו דנים ברעיון של איתות, שבו סוכנים משתפים ידע על ידי שינוי מצב העולם, וקראנו לאלגוריתם זה QDec-FPS. אנו גם דנים בתקפות, שלמות ותכונות תיאורטיות נוספות של שתי הגישות.

בכדי לבדוק את מאפייני הגישות שלנו, אנו ממדלים ומתארים מספר בעיות MAP חדשות, ואנו מראים אמפירית שמתכנן QDec-FP מתמודד הרבה יותר טוב ממתכנן ה- IMAP עבור בעיות MAP חדשות ומוכרות כאחד. נציין כי בעבר, IMAP היה האלגוריתם הטוב ביותר לפתרון QDec-POMDP, שהתמודד טוב משמעותית מאלגוריתמי Dec-POMDP עם בעיות גדולות. כתוצאה מכך, QDec-FP מאפשר פתרון בעיות מורכבות עוד יותר, הרבה מעבר לאלו שאלגוריתמי Dec-POMDP המתקדמים ביותר כיום יכולים לפתור. לאחר מכן, אנו בודקים את יכולת ההתמודדות של QDec-FPS עם בעיות גדולות, על ידי השוואה ל-QDec-FP ומראים כי ביצועי QDec-FPS טובים בהרבה ושימושיים יותר. יותר מכך אנו מראים כי גישה חדשה זו פותרת בעיות MAP שלא ניתנות לפתרון מעשי על ידי QDec-FP, מכיוון שהן דורשות שימוש באיתות.

אחד ממאפייני הליבה של בעיות MAP בהן השתמשנו לצורך הערכת מודל QDec-POMDP הוא שהן לעיתים ממדלות סוכנים הקשורים באופן רופף. בעיות MA אלו מניחות כי פעולות סוכן בודד אינן קשורות לפעולות סוכן אחר המבוצעות במקביל. או לחילופין, אם שתי פעולות של סוכן בודד או יותר מקיימות אינטראקציה, האינטראקציה ממודלת באמצעות פעולה שיתופית מפורשת המגדירה אותה.במובן מסוים, בעיות תכנון מרובות סוכנים אלו ממדלות פעולות מקביליות בצורה שימושית, אך ללא התחשבות בתכנון יעיל לפתרון הבעיה.

בבדי לחקור טוב יותר תיאורים יעילים של בעיות בהן קיימת אינטראקציה חזקה יותר בין פעולות ,נבחן בחלק הבא של העבודה פעולות עם אינטראקציה- פעולות שההשפעה המשותפת שלהן שונה מהחיבור של ההשפעות הנפרדות שלהן. פעולות אלה מאתגרות לייצוג ולתכנון בשל האופי הקומבינטורי שלהן .אנו בוחנים את ייצוג הפעולות הללו בהגדרת MAP הפשוטה ביותר, כאשר מצב העולם ידוע, ללא אי וודאות. עד כה נעשו מעט ניסיונות לספק שפה תמציתית לייצוגם שיכולה לתמוך גם בתכנון ריכוזי יעיל וגם בתכנון מבוזר השומר על פרטיות הסוכנים. אנו מציעים גישה לייצוג תמציתי של פעולות עם אינטראקציה ,ומראים כיצד מודל זה יכול להיות מומר לבעיית סוכן בודד כמו גם לבעיית ריבוי סוכנים השומרת על פרטיות הסוכנים. אנו בודקים את ביצועי השיטה שלנו בבמה בעיות חדשות הכוללות פעולות עם אינטראקציה וצורך בשמירה על פרטיות.