Collaborative Multi-Agent Planning with Black-Box Agents by Learning Action Models

Argaman Mordoch,[1] Daniel Portnoy,[1] Brendan Juba, [2] Roni Stern [1,3]

[1] Ben Gurion University
[2] Washington University
[3] Palo Alto Research Center
mordocha,portdan,sternron@post.bgu.ac.il, bjuba@wustl.edu

Abstract

We explore the problem of generating a plan for a team of heterogeneous collaborative agents without knowing their capabilities, but having access to observations of previously executed of plans. To plan for such "black-box" collaborative agents, we present the Planning using Offline Learning (POL) framework. POL compiles the given observations into trajectories of a single "super" agent, and uses an action model learning algorithm to learn the capabilities of that "super" agent. We implemented POL for Multi-agent STRIPS (MA-STRIPS) domains, and show that when using the Safe Action Model (SAM) learning algorithm, it is guaranteed to be sound and have a probabilistic form of completeness. Empirically, we evaluate POL over a standard MA-STRIPS benchmark. The results show that an almost perfect action model was learned for all agents with only a few trajectories in most cases. Finally, we discuss how POL and SAM learning can be extended to handle observations with concurrent and possibly conflicting actions.

Introduction

Many real-world applications require planning for a group of collaborative autonomous agents, e.g., in furniture assembly systems (Knepper et al. 2013), automated warehouses (Azadeh, De Koster, and Roy 2019), sensor networks (Lesser, Ortiz Jr, and Tambe 2003), and teams of robots performing search and rescue missions (Allouche and Boukhtouta 2010). Collaborative multi-agent planning (CMAP) is a well-known challenge in the Artificial Intelligence literature, and multiple formalisms have been proposed to represent CMAP problems and in particular the capabilities of the different agents (Brafman and Domshlak 2013; Bernstein et al. 2002; Tambe 1997). Yet obtaining a formal model of the capabilities of all agents in the team is often a difficult manual task. In this work, we focus on a CMAP setting in which the planning agent is not given a model of capabilities of the other agents. Such a setting may also occur in ad-hoc teamwork setups (Barrett and Stone 2012), where other agents' interfaces do not support sharing their internal models (Verma, Marpally, and Srivastava 2021), and where agents do not wish to share this information due to privacy reasons (Brafman and Domshlak 2013; Maliah, Shani, and Stern 2016). Generating a collaborative plan with such "black-box" agents is particularly challenging in mission-critical domains. In these domains, plan execution failures must be avoided, and thus trial-and-error approaches, which are common in the multi-agent reinforcement learning (MARL) literature, cannot be used.

Instead of knowing the other agents' capabilities, we assume access to a set of *trajectories*, i.e., sequences of alternating states and actions, performed in the same domain (possibly planning to achieve different goals). Such trajectories may have been generated by humans who know the agents' capabilities or via an interactive querying process, as outlined by Verma et al. (2021).

For using these trajectories to generate new plans, we propose the POL framework. POL comprises three stages: compiling, learning, and planning. In the compilation stage POL processes all the given trajectories of the multi-agent plans and transforms them to trajectories of a single "super" agent that has all the actions observed in $\mathcal{T}$. Then, in the learning phase, POL employs a learning algorithm to generate a single-agent action model that represents the planning agent's view of the capabilities of the agents in the team. Finally, in the planning phase, POL uses learned single-agent action model to generate plans by using an off-the-shelf single-agent planner. This framework is the first contribution of this work.

The second contribution of this work is a description and theoretical analysis of a POL implementation for CMAP problems expressed in Multi-agent STRIPS (MA-STRIPS) (Brafman and Domshlak 2013). MA-STRIPS is a CMAP formalism for agents that operate in a closed discrete world, have full observability, and their actions have deterministic effects. Coupled with the Safe *safe* Action Model (SAM) learning algorithm (Juba, Le, and Stern 2021; Stern and Juba 2017), POL can guarantee that every generated plan is compatible with the actual capabilities of all agents. Moreover, under certain assumptions, only a small number of trajectories are needed to guarantee that, with high probability, such a plan can be found for a given problem in the same lifted domain. The third contribution of this work is an experimental evaluation of POL on a standard MA-STRIPS benchmark, using several action model learning algorithms, namely *SAM* learning (Stern and Juba 2017), *ESAM* learning (Juba, Le, and Stern 2021), and FAMA (Aineto, Celorrio, and Onaindia 2019a). Our results show that using a small number of observations, POL can create multi-agent plans for almost all test problems. Finally, we describe how POL can be implemented for MA-STRIPS with concurrent, possibly conflicting actions. This requires extending SAM learning to this setting, which requires disambiguating the effects of actions that are performed concurrently and learning which actions can be performed in parallel.

Background and Problem Definition

STRIPS is arguably the most straightforward and well-known language for formalizing single-agent planning. It uses uses propositional logic to define the planning **domain** and **problem**, as follows:

Definition 1 (STRIPS)**.** A STRIPS problem is represented by a tuple $\Pi = \langle P, A, I, G\rangle$ where:

- P is a finite set of propositions.
- A is the set of actions the agent can perform.
- I is the initial state.
- G is the goal to achieve.

A proposition $p \in P$ describes a possible fact about the world. A state is a set of facts ($s \subseteq P$), representing that the conjunction of these facts are true and all other facts are not. A is the finite set of actions that the agent can perform. Each action a is defined by its preconditions (*pre*(a)) and effects (*eff*(a)). Preconditions and effects are sets of *literals*, where a literal is either a fact $p \in P$ or its negation. The effects of an action are often separated into add-effects and delete-effects, where a positive fact is an add-effect, and a negative fact is a delete effect. An action a is *applicable* in a state s if all its preconditions are satisfied in s. The result of applying a to a state s, denoted by $a(s)$, is a state that contains all the facts in *eff*(a) as well as all the facts in s except whose negations are in *eff*(a). The initial state I is a state and the goal G is a consistent set of literals. A state s *satisfies* a set of literals L iff s satisfies all the literals in L. A state s_G that satisfies the goal condition G is referred to as a goal state. A solution to the STRIPS planning task is a *plan*, which is a sequence of actions $(a_1, \ldots, a_k)$ such that (1) *I satisfies pre*(a_1), and (2) $a_k(\ldots(a_1(I)\ldots)$ *satisfies G*, i.e., a plan that can be applied to the initial state and results in a state that satisfies the goal.

Multi-agent STRIPS (MA-STRIPS) (Brafman and Domshlak 2008) is an extension of STRIPS that supports planning for multiple agents. MA-STRIPS generalizes STRIPS by defining a finite set of actions for each agent, characterizing the capabilities of that agent. The formal definition is as follows:

Definition 2 (MA-STRIPS)**.** A MA-STRIPS problem is represented by a tuple $\Pi = \langle P, k, \{A_i\}_{i=1}^{k}, I, G\rangle$ where:

- P, I and G are the set of propositions, initial state, and goal, respectively.
- k is the number of agents.
- A_i is the set of actions agent i can perform.

For simplicity, we assume agents perform their actions **sequentially** and not in parallel (we revisit this assumption later in the paper). Under this assumption, a solution to the MA-STRIPS planning task is a sequence of actions $(a_1, \ldots, a_k)$ that can be applied to I and results in a state that satisfies G. Each action in the plan is a member of the set of actions of one of the agents. For example, a solution to a MA-STRIPS problem can be a sequence of actions where the first agent performs the first two actions and the second agent performs the next four. Note that concurrency is not explicitly defined in the MA-STRIPS formalism, and to the best of our knowledge all MA-STRIPS planners output a sequential plan comprising a sequence of single agent actions. Recent extensions to formally define concurrent actions are not yet adopted in the CMAP community (Shekhar and Brafman 2020).

The main source of information we assume are *trajectories* collected by observing previously executed plans.

Definition 3 (Trajectory)**.** A trajectory $T = \langle s_0, a_1, s_1, \ldots a_n, s_n\rangle$ is an alternating sequence of states $(s_0, \ldots, s_n)$ and actions $(a_1, \ldots, a_n)$ that starts and ends with a state.

The trajectory created by applying π to a state s is the sequence $\left\langle s_0, a_1, \ldots, a_{|\pi|}, s_{|\pi|}\right\rangle$ such that $s_0 = s$ and for all $0 < i \leq |\pi|$, $s_i = a_i(s_{i-1})$. In prior work (Wang 1994, 1995; Walsh and Littman 2008; Stern and Juba 2017; Arora et al. 2018; Aineto, Celorrio, and Onaindia 2019b) a trajectory $\left\langle s_0, a_1, \ldots, a_{|\pi|}, s_{|\pi|}\right\rangle$ is often represented as a set of triples $\left\{ \langle s_{i-1}, a_i, s_i\rangle \right\}_{i=1}^{|\pi|}$. Each triplet $\langle s_{i-1}, a_i, s_i\rangle$ is called an *action triplet*, and the states s_{i-1} and s_i are referred to as the pre- and post- state of action a_i. We denote the set of all action triplets in the trajectories in $\mathcal{T}$ that include the grounded action a by $\mathcal{T}(a)$.

Finally, we can formally define the problem we consider in this work, which we refer to as the CMAP with black-box agents problem.

Definition 4 (CMAP with Black-Box Agents)**.** A CMAP with Black-Box Agents (CMAP-BB) problem is represented by a tuple $\langle \Pi, \mathcal{T}, i\rangle$ where:

- Π is a MA-STRIPS problem.
- $\mathcal{T}$ is a set of trajectories.
- i is an index of an agent specified in Π.

A solution to this CMAP-BB is a plan which is a solution to the underlying MA-STRIPS problem Π.

We refer to agent i as the *planning agent* and assume without loss of generality that i=0. The key constraint in solving a CMAP-BB problem is that the problem solver does not know the actions of any agent except that of the planning agent and the actions observed in the given trajectories ($\mathcal{T}$). In addition, the problem solver does not receive an *action model*, i.e., the preconditions and effects, of any action except for those of the planning agent. We denote the action model for all agents by M^* and refer to it as the *accurate* action model.

Since multi-agent problems vary significantly in their assumptions, we list the ones we make in this work. The agents are collaborative and do not aim to obfuscate their action

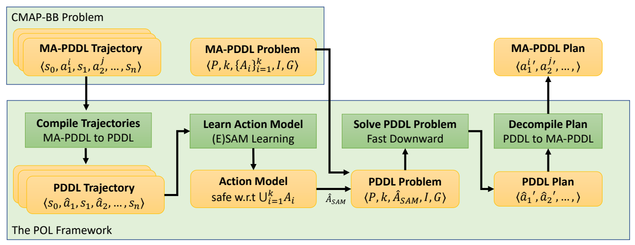
CMAP-BB Problem
MA-PDDL Trajectory
$\langle s_0, a_1^i, s_1, a_2^j, \ldots, s_n \rangle$
MA-PDDL Problem
$\langle P, k, \{A_i\}_{i=1}^k, I, G \rangle$
MA-PDDL Plan
$\langle a_1^i{}', a_2^j{}', \ldots, \rangle$
Compile Trajectories
MA-PDDL to PDDL
Learn Action Model
(E)SAM Learning
Solve PDDL Problem
Fast Downward
Decompile Plan
PDDL to MA-PDDL
PDDL Trajectory
$\langle s_0, \hat{a}_1, s_1, \hat{a}_2, \ldots, s_n \rangle$
Action Model
safe w.r.t $\cup_{i=1}^k A_i$
$\hat{A}_{SAM}$
PDDL Problem
$\langle P, k, \hat{A}_{SAM}, I, G \rangle$
PDDL Plan
$\langle \hat{a}_1{}', \hat{a}_2{}', \ldots, \rangle$
The POL Framework

Figure 1: A diagram of the POL framework for solving CMAP-BB problems.

models explicitly. The world is deterministic and fully observable, and the given trajectories provide a complete and accurate depiction of observed previously executed plans. When observing an action belonging to the non-planning agents in a trajectory, the planning agent only receives the signature of that action. While these assumptions are strong, they are commonly assumed in the automated planning literature and are also a helpful abstraction in many real-world applications.

The POL Framework

The approach we propose for solving a CMAP-BB problem, called POL, consists of three phases: compilation, learning, and planning. In the compilation phase, POL processes all the given trajectories of the multi-agent plans and transforms them to trajectories of a single "super" agent that has all the actions observed in $\mathcal{T}$. In the learning phase, POL applies a learning algorithm to learn an action model for that "super-agent's" actions based on the transformed trajectories. POL uses this learned action model to create a single-agent STRIPS problem corresponding to the given CMAP-BB problem. Finally, in the planning phase, POL uses a single-agent planner to generate a solution for that problem, which translates to a solution to the given multi-agent CMAP-BB problem. Figure 1 illustrates how POL is used to solve a CMAP-BB problem.

POL's compilation phase is pretty straightforward. It consists of iterating over every action in the given multi-agent trajectories $\mathcal{T}$ and creating a corresponding action for the "super" agent. The identity of the agent performing each action is inserted as a parameter of the super agent's action. For example, consider a trajectory from a multi-agent logistics problem with two truck agents $t1$ and $t2$ that includes the truck agent $t1$ performing `(load p1 l1)` and the truck agent $t2$ performing `(load p2 l2)`. POL will compile this multi-agent trajectory to a single-agent trajectory that includes the actions `(load t1 p1 l1)` and `(load t2 p1 l1)`.[1] The result of this compilation phase is a set of single-agent trajectories.

For POL's learning phase, any single-agent action model learning algorithm that accepts a set of trajectories can be used. FAMA (Aineto, Celorrio, and Onaindia 2019b), ARMS (Wu, Yang, and Jiang 2007), LOCM (Cresswell, McCluskey, and West 2013), and SAM learning (Juba, Le, and Stern 2021; Stern and Juba 2017) are examples of such learning algorithms. Similarly, any single agent planner can be used for the POL's planning phase. In our experiments, we used for this purpose FastDownward (FD) (Helmert 2006), a well-known state-of-the-art single-agent planner.

[1]Note that this assumes the planning agent knows that the load actions of all agents are the same. In cases where the agent cannot discern this the it will consider these actions as distint and may learn a different action model for them.

Learning Safe Action Models in POL

Without any restrictions on the learning and planning algorithms used for the learning and planning phases, it is not easy to guarantee the success of the plans generated by POL. Next, we propose several restrictions over the algorithms used in the POL planning and learning phases that provide such guarantees. To this end, we borrow the notion of a *safe action model* from Juba et al. (2021).

Definition 5 (Safe Action Model)**.** An action model M' is safe with respect to an action model M if for every state s and action a it holds that if a is applicable in s according to M' then (1) a is also applicable in s according to M, and (2) applying a to s results in the same state according to both action models.

As noted by Juba et al. (Juba, Le, and Stern 2021), a direct implication of Definition 5 is that if an action model M' is safe w.r.t. some other action model M, then any plan that is sound according to M' will also be sound according to M (Juba, Le, and Stern 2021). This observation allows us to provide a similar guarantee in our context.

Theorem 1 (Soundness)**.** Let $\Pi_{\mathcal{T}}$ be a CMAP-BB problem $\langle \Pi, \mathcal{T} \rangle$. In any implementation of POL in which (1) the learning phase returns an action model that is safe w.r.t. the action model of the underlying MA-STRIPS problem Π, and (2) the planning phase returns a plan that is sound w.r.t. the learned action model, it holds that if POL returns a plan, then that plan is a solution for $\Pi_{\mathcal{T}}$.

Proof. Let M and M' be the action models of the underlying MA-STRIPS problem and learned by the POL learning phase, respectively. If the plan returned by POL is sound w.r.t. M' then that plan must also be sound w.r.t. M since M' is safe w.r.t. M. $\square$

Thus, implementing POL with an action-model learning algorithm that returns a safe action model and a sound single-agent planner guarantees that it will never return a multi-agent plan that cannot be executed by one of the agents. Nevertheless, such a POL implementation is not necessarily *complete*, i.e., it might fail to find a solution to a given CMAP-BB problem even if the underlying MA-STRIPS problem is solvable. An extreme example of this is when $\mathcal{T}$ is empty, i.e., zero trajectories are available. In this case, POL would practically attempt to solve a multi-agent problem using only the actions of the planning agent. If the problem requires cooperation with any other agents, then POL would fail to find a solution. More generally, having too few trajectories in $\mathcal{T}$ may result in learning a safe action model that is overly restrictive or even lack actions that are critical for achieving the goal.

Implementing POL with *SAM* Learning

Next, we describe an implementation of POL that is based on using the *SAM* learning algorithm (Stern and Juba 2017; Juba, Le, and Stern 2021) for the POL learning phase. For completeness, we begin by providing a brief description of *SAM* learning. Initially, *SAM* learning creates an action model in which each action a has all literals as a precondition and none of the literals as an effect. Then, for each state transition $(s, a, s') \in \mathcal{T}(a)$ the algorithm removes literals from $pre(a)$ that do not appear in s, and adds to $eff(a)$ every literal that is s' but not in s.

SAM learning has several attractive properties. First, it guarantees that the action model it returns is safe with respect to the action model of the domain that generated the trajectories. Second, its runtime is polynomial in the number of trajectories, actions, and literals. Third, if future problems are drawn from the same distribution as those used to generate the given set of trajectories, then the number of trajectories required to learn an action model that, with high probability, enables solving most problems is only quasi-linear in the number of actions and facts in the real domain (Theorem 2 in (Stern and Juba 2017)).

Thus, this implementation of POL has the following properties. First, any solution returned is a sound multi-agent plan. Second, the runtime of the learning phase is polynomial. Third, if future problems are drawn from the same distribution as the one used to generate trajectories, then the number of trajectories needed to guarantee that POL will be able to solve a randomly drawn problem with high probability is only quasi-linear in the number of actions, facts, and agents.

CMAP-BB in Lifted Domains

The discussion so far was limited to *grounded domains*, where actions and facts are not parameterized. Planning and learning in grounded domains are often highly inefficient, where the number of actions can be exponentially large. Instead, classical planning domains and problems are almost always provided in a *lifted representation*, usually specified in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998). Similarly, existing MA-STRIPS benchmarks are also available in a lifted representation called MA-PDDL (Kovacs 2012). This section describes the CMAP-BB problem in such lifted domains and how POL can be adapted accordingly.

A lifted classical planning domain is defined in PDDL by a set of types T, lifted fluents $\mathcal{F}$, lifted actions A, and their corresponding action models. Lifted fluents and actions are parameterized versions of the facts and actions in STRIPS, where each parameter is associated with a type $t \in T$. For example, the lifted action `(move ?obj - object ?from - location ?to - location)` has three parameters, `?obj`, `?from`, and `?to`, associated with the types `object`, `location`, and `location`, respectively. Similarly, the lifted fluent `(at ?obj - object ?loc - location)` has two parameters of types `object` and `location`. The action model of a lifted action a is its preconditions and effects, which are specified as lifted literals coupled with a binding function that maps the lifted fluent's parameters to the lifted action's parameters. A problem in PDDL specifies a set of objects O, each associated with a type from T. A *grounded action* and a *grounded fluent* are a lifted action and a lifted fluents, respectively, coupled with a binding function that maps their parameters to the objects specified in a PDDL problem. A state in PDDL is a set of grounded fluents. Similarly, the goal in PDDL is specified as a set of grounded literals. The desired solution, i.e., the plan, is a sequence of grounded actions.

MA-PDDL (Kovacs 2012) is a lifted multi-agent planning formalism that generalizes MA-STRIPS in a similar way, where agents' actions are defined in a lifted parametrized representation. The CMAP-BB problem definition can be naturally extended to this lifted formalism. The available trajectories $\mathcal{T}$ are sequences of grounded actions and states. The planning agent receives a set of trajectories $\mathcal{T}$, which are sequences of grounded actions and states, as well as the current PDDL problem Π, which includes the available objects, initial state, and goal. It does not know, of course, the lifted action model of the other agents.

POL for Lifted Domains

The POL framework is also applicable for this type of CMAP-BB problem but requires learning and planning algorithms that support lifted domains. Most modern planning algorithms provide such support.[2] Recently, the *SAM* learning algorithm has also been extended to support learning *lifted* safe action models (Juba, Le, and Stern 2021). This lifted version of *SAM* learning, however, can only consider action triplets in which each of the bindings of action parameters to objects is injective, i.e., a single object cannot be mapped to more than one parameter of the same action (Juba, Le, and Stern 2021). *SAM* learning cannot use action triplets in which this assumption, called the *injective action binding assumption* does not hold, without compromising its safety guarantee. Thus, it ignores such action triplets. If such triplets are common in the available trajectories, *SAM* learning's sample efficiency would be negatively affected. To learn safe action models in this setting, Juba et al. (2021) created the *ESAM* algorithm. This algorithm generates a CNF for each action it observes in the trajectory, representing its knowledge about which lifted literals may be preconditions and effect of that action. This CNF translates to a safe action model. In some cases, this CNF indicates the existence of ambiguity in the effects of an action which prevents directly creating a safe action model for that action. In such cases, *ESAM* creates *proxy-actions* that represent specific forms of that action that can still be performed safely. The details of this algorithm are somewhat involved and are presented more clearly in Juba et al. (2021).

Theoretical Properties

POL with *ESAM* and a sound and complete single-agent planner has similar properties to POL for grounded domains. The multi-agent plans it returns are applicable by all agents

[2]Although the standard approach is to fully ground the domain as a preprocessing step.

(soundness), but it may not find a plan even for cases where such exists (incompleteness). Here too, the probability that will occur decreases quickly with the number of trajectories (probably approximately complete). More accurately, let $\mathcal{P}_D$ be a probability distribution over solvable planning problems in a domain D. Let $\mathcal{T}_D$ be a probability distribution over pairs $\langle P, T\rangle$ given by drawing a problem P from $\mathcal{P}(D)$, using a sound and complete planner to generate a plan for P, and setting T to be the trajectory from following this plan. Let $arity(a, t)$ and $arity(F, t)$ be the number of parameters in the lifted action and fluent a and F, respectively, of type t. The following result is proven by Juba et al. (2021) for the single-agent case and directly applies to POL as well.

Theorem 2. Given

$$m \geq \frac{1}{\epsilon}(2 \ln 3 \sum_{\substack{F \in \mathcal{F} \\ a \in \bigcup_{i=1}^{k} A_i}} \prod_{t \in T} arity(a,t)^{arity(F,t)} + \ln\frac{1}{\delta}) \quad (1)$$

trajectories sampled from $\mathcal{T}_D$, with probability at least $1-\delta$ the probability that a problem drawn from $\mathcal{P}_D$ will not be solvable by POL is at most ϵ.

Note that increasing the number of agents only increases the complexity linearly. To calculate the time complexity for POL we divide the calculation to three parts: First, the compilation stage consists of iterating over the actions in the multi-agent trajectories and creating the appropriate action for the "super" agent trajectory. This process is linear in the number of actions that were compiled in this stage. Second, The learning stage consists of executing *SAM*/ *ESAM* on the compiled trajectory and learning the "super" agent's action model. The complexity for this stage depends on whether or not proxy actions were created. In case proxy actions were not created, the time complexity for the learning stage of POL is linear in the number of triplets Juba, Le, and Stern (2021). On the other hand, if POL using *ESAM* needs to compile proxy actions the time complexity can rise up to be exponential. Finally, the planning stage's time complexity is the same as the complexity of the planner that is being used.

Experimental Results

We implemented POL for lifted domains using three different action-learning algorithms: *SAM* learning, *ESAM* learning (Juba, Le, and Stern 2021), and FAMA (Aineto, Celorrio, and Onaindia 2019a). Since existing MA-STRIPS planners do not generate plans with concurrent actions, we did not implement the support for concurrent actions mentioned above. We evaluated our POL implementations on the publicly available CoDMAP benchmark of MA-PDDL problems.[3] This benchmark includes 10 MA-PDDL domains and 20 MA-PDDL problems for each domain. Table 1 shows general statistics on the selected domains. The columns $|A|$ and $|P|$ list the total number of **different** actions and fluents in each domain. The columns "Act." and "Flu." list each domain's maximal arity for actions and fluents. The column "I.B.A" indicates whether or not the trajectories in our benchmark maintained the injective binding assumption required for *SAM*.

[3]http://agents.fel.cvut.cz/codmap/

			Arity		
Domain	$\|A\|$	$\|P\|$	**Act.**	**Flu.**	**I.B.A.**
blocks	4	5	3	2	yes
depot	5	7	4	3	yes
driverlog	6	6	4	2	yes
elevators	6	8	5	2	no
logistics	5	3	4	3	yes
rovers	9	25	6	3	no
satellites	5	8	4	2	yes
taxi	17	6	2	2	yes
woodworking	13	14	9	3	no
zenotravel	5	4	6	2	yes

Table 1: Statistics of the maximal values for each of the tested domains.

Implementation Details

Since there is no publicly available implementation of *ESAM*, we implemented both *SAM* and *ESAM* learning algorithms in Python for use in the POL learning phase. Due to its complexity, we implemented a partial version of *ESAM* in which only some of ESAM's proxy actions are created. Specifically, *ESAM* resolves ambiguity about the possible effects of an action by creating two types of proxy actions: one that merges action parameters that are mapped to the same object into a single parameter and one that assumes imposes an additional precondition. We only implemented the latter approach as it was simpler technically and sufficient to solve all the problems in the available benchmark.

In addition, some domains contain *constants*, which are objects that are defined at the domain level and are present in all problems from that domain. The addition of constants to the domain definition adds complexity to the learning process since they extend the current set of objects to which the action and predicate parameters can be mapped. Indeed, neither *SAM* nor *ESAM* directly support constants, as they assume the parameters of an actions' preconditions and effects can only be bound to parameters of the action. Therefore, we extended both algorithms to support constants by allowing such binding of literals to constants. The impact of this addition on the sample complexity analysis given in Theorem 2 is reasonable, adding the number of constants in the domain to the base of the exponent in Equation 1. Practically, we limited the scope of learning by assuming that a grounded literal ℓ is a precondition or effect of a grounded action a only if they share at least one object in their parameters. This prevents having lifted literals with only constants as preconditions or effects and improves the learning efficiency. We also verified this assumption empirically in all our benchmark problems.

To implement POL with FAMA, we used a publicly available implementation of FAMA.[4] FAMA runs an internal planner to generate its action model. As recommended by FAMA, we used the Madagascar single-agent planner (Rintanen 2014) for this purpose, setting a time limit of 100 seconds. In preliminary experiments, we observed that indeed

[4]https://github.com/daineto/meta-planning

Madagscar is well-suited for this task, and increasing the time limit beyond 100 seconds did not yield significant benefits. Also, we note that the woodworking and taxi domains included constants, which FAMA does not support, and thus we did not include FAMA results for these domains.

In all our POL implementations, we used the FD planner for the POL planning phase, with a time limit of 60 seconds. Since we do not aim for optimal solutions, we configured FD to use Greedy Best-First Search with the FF heuristic and preferred operators.

Evaluation Setup

Since the number of problems for each domain is relatively small in the available benchmarks, we evaluated our algorithms using the k-fold cross-validation method (Refaeilzadeh, Tang, and Liu 2009). Specifically, we split our dataset into five disjoint folds, each comprising four problems. Thus, in each fold, 16 problems were used to generate the train set trajectories, and the remaining four problems were used to test the learned action model. These trajectories were generated by converting the 16 MA-PDDL problems into single-agent problems and running Fast Downward (Helmert 2006), an off-the-shelf state-of-the-art planner, to solve them, using a time limit of 1 minute.[5] In a few cases, FD could not solve all 16 problems. This occurred only in depot and driverlog domains. In these cases, fewer trajectories were obtained and used for training.

We focused our evaluation on three metrics: the number of problems solved with POL, the precision of the learned action model, and the recall of the learned action model, denoted S, P, and R, respectively. The S metric is computed by running POL on the problems in our test set. Note that if POL outputs a plan that is not sound, then it is not counted as a solved problem. Precision and recall (P and R) are measured separately for preconditions, add- and delete effects, and denoted P_{add}, P_{del}, R_{add}, and R_{del}.

Experimental Results

While conducting our experiments, we noticed that FAMA could only generate action models for the blocks, driverlog, logistics, and zenotravel domains before running out of time or memory. In these domains, the injective binding assumption always holds (see Table 1). Thus, the behavior of POL with either *SAM* or *ESAM* is the same, and we only report the results for POL with *SAM* learning. These results are presented in Table 2. The column "Alg." indicates the learning algorithm used, FAMA or *SAM*, denoted in the table as F and S, respectively. The columns $\mathcal{T}$ and "Tri." indicate the number of trajectories and action triplets needed to obtain the best results. Here, best results mean maximizing S, and then maximizing the precision and recall results. The rest of the columns show the minimum, average, and maximum across all folds for all our metrics. Observe that while *SAM* learning is monotonic, in the sense that adding more trajectories can only increase its performance (P, R, and S values), this is not the case for FAMA.

[5]Increasing the FD time limit did not significantly increase the number of problems solved.

As can be seen, POL with *SAM* can always solve the same or more problems than when using FAMA. The advantage of POL with *SAM* is evident in the driverlog and zenotravel domains, where POL with *SAM* solved all 4 test problems while POL with FAMA solved only one or zero problems, respectively. In both cases, POL with FAMA generated multi-agent plans, but these plans were inapplicable (i.e., not sound). Previous work (Verma, Marpally, and Srivastava 2021) also encountered similar results.

In terms of precision and recall, the action model learned using *SAM* always yielded the same or higher precision and recall compared to FAMA. Observe that for driverlog, and zenotravel domains, POL with FAMA, peaked after fewer action triplets than *SAM*. However, in these cases, its peak performance was significantly lower than POL with *SAM*, solving fewer problems in the test set (lower S values) and lower precision and recall results. Notably, the precision and recall computation for FAMA was different than as we computed. In FAMA, the precision and recall are averaged over the TP, FP, and FN of all actions, while we computed the precision and recall per action and reported the average of these values. Nevertheless, in all cases except driverlog *SAM* was able to learn precisely the real action model, and thus these different computations are not significant.

In general, the results highlight that for CMAP-BB, POL with *SAM* performs significantly better than POL with FAMA. However, it is essential to note that the primary purpose of FAMA is learning action models in a *partially observable* environment while our algorithm only works in *fully observable* ones. This property of FAMA correlates with the fact that its best performance results were acquired after a few trajectory triplets. Furthermore, we noticed that while POL with *SAM* learning only deduced the preconditions and effects for observed actions, FAMA also partially learned unobserved actions.

Results over the Entire Benchmark Unlike FAMA, both *SAM* and *ESAM* feature feasible time complexity. Thus, we were able to run POL with *SAM* and with *ESAM* over our entire benchmark. The results are reported in Figure 2. The table in Figure 2 (left) is in the same format as Table 2. Since both *SAM* and *ESAM* return a safe action model, the preconditions' recall and the add and delete effects' precision values are constantly 1.0. Thus, we omitted these columns from the table. The table shows only the results for POL with *SAM* learning since we observed that POL with *ESAM* performed the same in almost all cases. The only difference observed between *SAM* and *ESAM* manifested in the satellite domain and only in the number of triplets needed to reach the best performance. We report these results in the plot in Figure 2 (right), which shows the number of triplets needed to reach the best performance in each of our five folds. This similarity in performance between *SAM* and *ESAM* is expected since the injective binding assumption holds in all domains except elevators, rovers, and woodworking. Moreover, even in these domains, cases where this assumption does not hold are relatively rare.

The first trend we observe is that the small number of trajectories used for learning was sufficient to solve all prob-

Domain	Alg.	$\lvert\mathcal{T}\rvert$	Tri.	P_{pre}	R_{pre}	P_{add}	R_{add}	P_{del}	R_{del}	S
blocks	S	1, 1, 1	38, 38, 40	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**4, 4, 4**
	F	1, 1, 1	38, 38, 40	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**4, 4, 4**
driverlog	S	9, 10, 11	143, 183, 249	**0.9, 0.9, 0.9**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	4, 4, 4
	F	1, 3, 4	14, 34, 57	0.7, 0.8, 0.9	0.6, 0.7, 0.8	0.4, 0.6, 0.7	0.7, 0.8, 1.0	0.7, 0.8, 1.0	0.9, 0.9, 1.0	0, 0, 1
logistics	S	1, 1, 1	46, 47, 49	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**4, 4, 4**
	F	1, 2, 2	49, 81, 89	**1.0, 1.0, 1.0**	0.8, 0.8, 0.9	0.9, 0.9, 1.0	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	0, 1, 4
zenotravel	S	2, 3, 3	55, 72, 82	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**1.0, 1.0, 1.0**	**4, 4, 4**
	F	1, 1, 1	24, 26, 27	0.6, 0.6, 0.7	0.6, 0.7, 0.7	0.4, 0.6, 0.7	0.7, 0.8, 0.9	0.6, 0.6, 0.7	0.7, 0.7, 0.9	0, 0, 0

Table 2: Comparison: POL using FAMA and *SAM*. The best results in each case is given in bold.

Domain	$\mathcal{T}$	Tri.	P_{pre}	R_{add}	R_{del}	S
blocks	1, 1, 1	38, 38, 40	1.0, 1.0, 1.0	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4
depot	1, 1, 1	31, 201, 243	0.9, 0.9, 0.9	1.0, 1.0, 1.0	1.0, 1.0, 1.0	3, 4, 4
driverlog	9, 10, 11	159, 183, 249	0.9, 0.9, 0.9	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4
logistics	1, 1, 1	46, 47, 49	1.0, 1.0, 1.0	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4
satellite	1, 4, 6	40, 229, 324	0.8, 0.9, 1.0	0.8, 0.9, 1.0	0.7, 0.8, 0.9	3, 4, 4
taxi	16, 16, 16	349, 367, 405	0.8, 0.9, 0.9	1.0, 1.0, 1.0	1.0, 1.0, 1.0	3, 4, 4
zenotravel	2, 3, 3	55, 72, 82	1.0, 1.0, 1.0	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4
rover	4, 6, 8	281, 352, 409	0.8, 0.8, 0.8	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4
woodworking	6, 7, 11	260, 320, 447	0.6, 0.6, 0.6	0.6, 0.6, 0.6	0.6, 0.6, 0.6	4, 4, 4
elevators	2, 4, 5	148, 326, 400	0.7, 0.7, 0.7	1.0, 1.0, 1.0	1.0, 1.0, 1.0	4, 4, 4

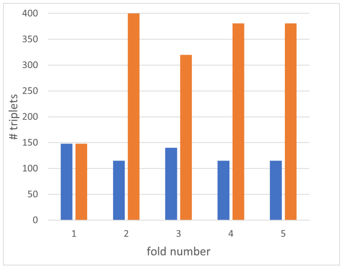

Figure 2: (Left) Results for POL with *SAM* learning. (Right) Comparison of the number of triplets needed until complete learning for both *SAM* (marked in blue) and *ESAM* (marked in orange) in the elevators domain.

lems in our test set in almost all cases. Moreover, for the blocks, depot, logistics, and zenotravel domains, the learning converged after less than 100 action triplets. For the *taxi* domain, our algorithm required all 16 trajectories in the training set to learn a model that is adequate to solve the test set problems. This happened since it is divided so that some agents are not observed. Thus, their actions could not be learned until the entire training set was acquired. There were a few cases where not all of the test set problems were solved. For instance, in the *depot* domain, for one fold, only three out of the four test set problems were solved. We performed a deeper investigation of these few cases and discovered that they occur either when the problem itself is too complicated for our planner to solve, even with the actual action model (this was the case in depot), or when an action has not been observed at all in the training set trajectories (this was the case in satellite).

The second trend we observe is that in all domains except woodworking and satellite, our algorithm can learn the action model of all agents almost perfectly, with a recall of 1.0 for the effects and an average precision higher than 0.7 in all cases (and usually much higher). The satellite domain presented an interesting phenomenon: the minimal recall for effects occurred when an action was not observed in a specific fold. Indeed, without observing an action, our algorithm cannot learn it. When experimenting on the woodworking domain, we discovered that to solve the test set problems, it was unnecessary to learn all of the actions and that using 7 out of the 12 actions, all of the test set problems were indeed solved. Thus, while the correct action model was not accurately learned, a sufficient, safe action model was found.

Finally, consider the difference between the results of *SAM* and *ESAM* on the elevators domain, as shown in Figure 2 (right). As can be seen, in this domain, *ESAM* was able to reach peak performance much faster than *SAM*, i.e., with significantly fewer action triplets. On average, *ESAM* converged in this domain after less than half of the triplets needed for *SAM* to converge. We note that the minimal number of triplets needed for *SAM* to converge for the domain is as high as the maximal number of triplets needed for *ESAM*. To the best of our knowledge, this is the first experimental evidence for the benefit of using *ESAM* over *SAM* learning.

Concurrent Actions

If more than one agent can act at the same time, then a plan comprises joint actions, where a joint action represents at most a single action per agent. Formally, a joint action is a k-dimensional vector $\hat{a}$ where entry i in that vector, denoted $\hat{a}[i]$, is either an action from A_i or $\perp$, where the $\perp$ sign indicates that agent i does not perform an action in the joint action $\hat{a}$. Correspondingly, a trajectory is an alternating sequence of states and joint actions.

It is possible to extend the POL framework to support such settings with concurrent actions as follows. The "super" agent in POL's compilation phase is now capable of performing joint actions (as opposed to single-agent actions). The learning phase applies an action model learning algorithm over these trajectories of joint actions, explicitly learning an action model for the joint actions. That is, this action model explicitly specifies the preconditions and effects of each joint action independently.

The safety property of POL is preserved — plans generated with this joint action model are guaranteed to be sound. Yet explicitly learning a joint action model is highly inefficient, as the number of actions in this action model grows exponentially with the number of agents. To achieve the form of approximate completeness described in Theorem 2 requires observing a number of trajectories that is exponential in the number of agents, as opposed to only linear in the number of agents, as is the case in the sequential case.

Independent Actions

The above challenge can be alleviated by making assumptions about the factored nature of a multi-agent planning problem. For example, consider the following assumptions, referred to as the *independent actions assumption*. Under this assumption, each single-agent action has its own preconditions and effects, and: (1) an action a_i can be applied if and only if its preconditions hold, regardless of the actions performed concurrently by the other agents; and (2) the effects of any joint action that includes a_i includes all of the effects of a_i. To the best of our knowledge, all MA-STRIPS planners implicitly make the independent actions assumptions, perhaps because conflicting effects are not well-defined in MA-STRIPS. Under the independent actions assumption, we propose the following version of SAM learning.

Definition 6 (SAM rules for independent concurrent actions)**.** For any observed action triplet $\langle s, \hat{a}, s' \rangle$

1. If $\ell \notin s$ then ℓ is not a precondition of any action $a \in \hat{a}$.
2. If $\ell \notin s'$ then ℓ is not an effect of any action $a \in \hat{a}$.
3. If $\ell \in s' \setminus s$ then $\exists a \in \hat{a}$ in which ℓ is an effect.

These learning rules form the basis for SAM learning for independent concurrent actions, in the same way that similar rules form the basis of *ESAM*. Initially, we assume all literals are preconditions of all single-agent actions, and the effects of all single-agent actions are empty. Then, we remove preconditions and add effects for the different single-agent actions by processing all the trajectories and applying the learning rules above. Note the third learning rule may create a disjunction over the possible effects of an action. The resulting action model can then be applied safely, for all states where these disjunctions allow determining safely the effects. This can be handled by replacing an action that has potentially ambiguous effects with proxy actions that ensure no ambiguity of effects exists. For more details, see how such proxy actions are created and used by the *ESAM* algorithm (Juba, Le, and Stern 2021).

Possibly Non-Independent Concurrent Actions

Consider two truck agents in the logistics domain, trying to load the same package in the same joint action. The effect of each load action independently is to have the package loaded onto the loading track, yet the effect of performing a joint action with both load actions cannot be that both trucks are loaded with the same package.

Without any additional assumption about the relation between the preconditions and effects of the joint action in those of its constituent actions, not much can be done beyond explicitly learning the joint action model. Consider the following relaxed form of the independent actions assumption: actions are independent except that performing some pairs of actions concurrently is forbidden. We refer to such a pair of actions as a *conflicting* pair of actions. Under this assumption, we can augment the rules in Definition 6 by adding that a pair of actions a and a' can only be performed concurrently, i.e., do not conflict, if there have been observed in the same joint action in the given observed trajectories. Adding this restriction maintains safety, but limits the ability to generalize, and increase the number of trajectories needed to guarantee our probabilistic form of completeness.

A more relaxed assumption is that a pair of actions cannot be performed concurrently iff they have conflicting effects. That is, a pair of single-agent actions a and a' are in conflict, denoted *conflict*(a, a'), if there exists a literal ℓ such that $\ell \in$ *eff*$(a) \wedge \ell \in$ *eff*(a'). A joint action $\hat{a}$ is said to have a conflict if it has constituent single-agent actions that are in conflict. This natural definition of conflicts corresponds, for example, to our logistics example with trucks picking up the same package. Namely, the actions $load - truck$ and $unload - truck$ are in conflict since both affect the same literals. If the observed trajectories contain only joint actions that do have a conflict, then we can extend the *SAM* rules in Definition 6 with the following rule: for every observed action triplet $\langle s, \hat{a}, s' \rangle$, action pair $a, a' \in \hat{a}$:

$$\forall \ell \in L : \bigwedge_{a, a' \in \hat{a}} \ell \notin eff(a) \vee \ell \notin eff(a') \tag{2}$$

Actions can be be performed concurrently safely if their preconditions are consistent with each other and their effects do not share any literal. This can safely be done using the action model returned by SAM learning.

Conclusions and Future Work

This paper introduced the CMAP-BB problem, where an agent is tasked to generate a multi-agent plan for a team of black-box agents. We proposed POL, a framework for solving CMAP-BB problems that learns an explicit action model for the agents in the team. Equipped with a Safe Action Model learning algorithm, POL is guaranteed to return a sound plan and, given enough trajectories, is probabilistically complete. We implemented POL for CMAP problems specified in MA-STRIPS, and evaluated it using three different action model learning algorithms. Our results, over 10 benchmark domains, showed that using a small number of trajectories is sufficient for POL to learn an action model that serves as an adequate approximation of the actual action model that enables producing applicable plans for most test problems. Comparing the different learning algorithms within POL in a fully observable setting showed the benefit of using a safe action model learning algorithm, namely *SAM* learning, over FAMA, a state-of-the-art action-learning algorithm. Finally, we discuss how POL and *SAM* learning can be extended to support problems in which agents can act concurrently, and their actions can potentially conflict. Future work will connect this version of *SAM* learning to recent progress in modeling concurrent actions (Shekhar and Brafman 2020), as well as explore how to implement POL in the context of richer planning models that include stochasticity, partial observability, and numerical state variables.

References

Aineto, D.; Celorrio, S.; and Onaindia, E. 2019a. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019b. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Allouche, M. K.; and Boukhtouta, A. 2010. Multi-agent coordination by temporal plan fusion: Application to combat search and rescue. *Information Fusion*, 11(3): 220–232.

Arora, A.; Fiorino, H.; Pellier, D.; Etivier, M.; and Pesty, S. 2018. A review of learning planning action models. *Knowledge Engineering Review*, 33.

Azadeh, K.; De Koster, R.; and Roy, D. 2019. Robotized and automated warehouse systems: Review and recent developments. *Transportation Science*, 53(4): 917–945.

Barrett, S.; and Stone, P. 2012. An analysis framework for ad hoc teamwork tasks. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 357–364.

Bernstein, D. S.; Givan, R.; Immerman, N.; and Zilberstein, S. 2002. The complexity of decentralized control of Markov decision processes. *Mathematics of operations research*, 27(4): 819–840.

Brafman, R. I.; and Domshlak, C. 2008. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *ICAPS*, 28–35.

Brafman, R. I.; and Domshlak, C. 2013. On the complexity of planning for agent teams and its implications for single agent planning. *Artificial Intelligence*, 198: 52–71.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(2): 195–213.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 379–389.

Knepper, R. A.; Layton, T.; Romanishin, J.; and Rus, D. 2013. Ikeabot: An autonomous multi-robot coordinated furniture assembly system. In *2013 IEEE International conference on robotics and automation*, 855–862. IEEE.

Kovacs, D. L. 2012. A Multi-Agent Extension of PDDL3.1. In *Workshop on the International Planning Competition (IPC) in the International Conference on Automated Planning and Scheduling (ICAPS)*, 19–27.

Lesser, V.; Ortiz Jr, C. L.; and Tambe, M. 2003. *Distributed sensor networks: A multiagent perspective*, volume 9. Springer Science & Business Media.

Maliah, S.; Shani, G.; and Stern, R. 2016. Collaborative privacy preserving multi-agent planning. *Autonomous Agents and Multi-Agent Systems*, 1–38.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language.

Refaeilzadeh, P.; Tang, L.; and Liu, H. 2009. Cross-validation. *Encyclopedia of database systems*, 5: 532–538.

Rintanen, J. 2014. Madagascar: Scalable planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21: 1–5.

Shekhar, S.; and Brafman, R. I. 2020. Representing and planning with interacting actions and privacy. *Artificial Intelligence*, 278: 103200.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *the International Joint Conference on Artificial Intelligence (IJCAI)*, 4405–4411.

Tambe, M. 1997. Towards flexible teamwork. *Journal of artificial intelligence research*, 7: 83–124.

Verma, P.; Marpally, S. R.; and Srivastava, S. 2021. Asking the right questions: Interpretable action model learning using query-answering. In *AAAI*.

Walsh, T. J.; and Littman, M. L. 2008. Efficient learning of action schemas and web-service descriptions. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 8, 714–719.

Wang, X. 1994. Learning planning operators by observation and practice. In *Second International Conference on Artificial Intelligence Planning Systems (AIPS)*, 335–340.

Wang, X. 1995. Learning by observation and practice: an incremental approach for planning operator acquisition. In *International Conference on Machine Learning (ICML)*, 549–557.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review*, 22(2): 135–152.