The 30th International Conference on

Automated Planning and Scheduling

DMAP 2020

Proceedings of the 6th Workshop on Distributed and Multi-Agent Planning

Edited by: Ronen I. Brafman, Eva Onaindia, and Shashank Shekhar

List of Papers

A Feedback Scheme to Reorder a Multi-Agent Execution Schedule by Persistently Optimizing a Switchable Action Dependency Graph

Alexander Berndt* **Niels van Duijkeren†** **Luigi Palmieri†** **Tamás Keviczky***

Abstract

In this paper we consider multiple Automated Guided Vehicles (AGVs) navigating a common workspace to fulfill various intralogistics tasks, typically formulated as the Multi-Agent Path Finding (MAPF) problem. To keep plan execution deadlock-free, one approach is to construct an Action Dependency Graph (ADG) which encodes the ordering of AGVs as they proceed along their routes. Using this method, delayed AGVs occasionally require others to wait for them at intersections, thereby affecting the plan execution efficiency. If the workspace is shared by dynamic obstacles such as humans or third party robots, AGVs can experience large delays. A common mitigation approach is to re-solve the MAPF using the current, delayed AGV positions. However, solving the MAPF is time-consuming, making this approach inefficient, especially for large AGV teams. In this work, we present an online method to repeatedly modify a given acyclic ADG to minimize the cumulative AGV route completion times. Our approach persistently maintains an acyclic ADG, necessary for deadlock-free plan execution. We evaluate the approach by considering simulations with random disturbances on the execution and show faster route completion times compared to the baseline ADG-based execution management approach.

Index terms— *Robust Plan Execution, Scheduling and Coordination, Mixed Integer Programming, Multi-Agent Path Finding, Factory Automation*

1 Introduction

Multiple Automated Guided Vehicles (AGVs) have shown to be capable of efficiently performing intra-logistics tasks such as moving inventory in distribution centers (Wurman, D'Andrea, and Mountz 2008). The coordination of AGVs in shared environments is typically formulated as the Multi-Agent Path Finding (MAPF) problem, which has been shown to be NP-Hard (Yu and LaValle 2012). The problem is to find trajectories for each AGV along a roadmap such that each AGV reaches its goal without colliding with the other AGVs, while minimizing the makespan. The MAPF problem typically considers an abstraction of the workspace to a graph where vertices represent spatial locations and edges pathways connecting two locations.

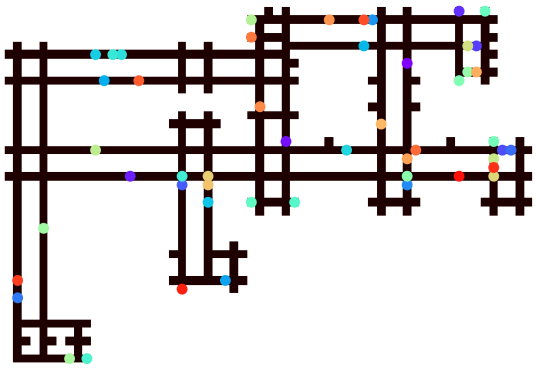


Figure 1: A roadmap occupied by 50 AGVs (represented by colored dots). AGVs must efficiently navigate from a start to a goal position while avoiding collisions with one another, despite being subjected to delays.

Recently, solving the MAPF problem has garnered widespread attention (Stern et al. 2019; Felner et al. 2017). This is mostly due to the abundance of application domains, such as intralogistics, airport taxi scheduling (Morris et al. 2016) and computer games (Ontanón et al. 2013). Solutions to the MAPF problem include Conflict-Based Search (CBS) (Sharon et al. 2015), Prioritized Planning using Safe Interval Path Planning (SIPP) (Yakovlev and Andreychuk 2017), declarative optimization approaches using answer set programming (Bogatarkan, Patoglu, and Erdem 2019), heuristic-guided coordination (Pecora et al. 2018) and graph-flow optimization approaches (Yu and LaValle 2013).

Algorithms such as CBS have been improved by exploiting properties such as geometric symmetry (Li et al. 2019),

*Alexander Berndt and Tamás Keviczky are with the Delft Center for Systems and Control (DCSC), TU Delft, 2628 CN Delft, The Netherlands berndtae@gmail.com, T.Keviczky@tudelft.nl.

†Niels van Duijkeren and Luigi Palmieri are with Robert Bosch GmbH, Corporate Research, Renningen, 71272, Germany {Niels.vanDuijkeren, Luigi.Palmieri}@de.bosch.com.

using purpose-built heuristics (Felner et al. 2018), or adopting a Mixed-Integer Linear Program (MILP) formulation where a branch-cut-and-price solver is used to yield significantly faster solution times (Lam et al. 2019).

Similarly, the development of bounded sub-optimal solvers such as Enhanced Conflict-Based Search (ECBS) (Barer et al. 2014) have further improved planning performance for higher dimensional state spaces. Continuous Conflict-Based Search (CCBS) can be used to determine MAPF plans for more realistic roadmap layouts (Andreychuk et al. 2019). As opposed to CBS, CCBS considers a weighted graph and continuous time intervals to describe collision avoidance constraints, albeit with increased solution times.

The abstraction of the MAPF to a graph search problem means that executing the MAPF plans requires monitoring of the assumptions made during the planning stage to ensure and maintain their validity. This is because irregularities such as vehicle dynamics and unpredictable delays influence plan execution. kR-MAPF addresses this by permitting delays up to a duration of k time-steps (Atzmon et al. 2020). Stochastic AGV delay distributions are considered in (Ma, Kumar, and Koenig 2017), where the MAPF is solved by minimizing the expected overall delay. These robust MAPF formulations and solutions inevitably result in more conservative plans compared to their nominal counterparts.

An Action Dependency Graph (ADG) encodes the ordering between AGVs as well as their kinematic constraints in a post-processing step after solving the MAPF (Hönig et al. 2017). Combined with an execution management approach, this allows AGVs to execute MAPF plans successfully despite kinematic constraints and unforeseen delays. This work was extended to allow for persistent re-planning (Hönig et al. 2019).

The aforementioned plan execution solutions in (Hönig et al. 2019; Atzmon et al. 2020; Ma, Kumar, and Koenig 2017) address the effects of delays by ensuring synchronous behavior among AGVs while maintaining the originally planned schedule's ordering. The result is that plan execution is unnecessarily inefficient when a single AGV is largely delayed and others are on schedule, since AGVs need to wait for the delayed AGV before continuing their plans. We observe that to efficiently mitigate the effects of large delays the plans should be adjusted continuously in an online fashion, where the main challenges are to maintain the original plan's deadlock- and collision-free guarantees.

In this paper, we present such an online approach capable of reordering AGVs based on a MAPF solution, allowing for efficient MAPF plan execution despite AGVs being subjected to large delays. This approach is fundamentally different from the aforementioned approaches (Hönig et al. 2017; Ma, Kumar, and Koenig 2017; Atzmon et al. 2020; Hönig et al. 2019) in that delays can be accounted for as they occur, instead of anticipating them *a priori*. The feedback nature of our approach additionally means solving the initial MAPF can be done assuming nominal plan execution, as opposed to solving a robust formulation which necessarily results in plans of longer length due to the increased conservativeness.

Our contributions include an optimization formulation based on a novel Switchable Action Dependency Graph (SADG) to re-order AGV dependencies. Monte-Carlo simulation results show lower cumulative route completion times with real-time applicable optimization times while guaranteeing collision- and deadlock-free plan execution.

Working towards our proposed solution, we formally define the MAPF problem and the concept of an ADG in Section 2. Based on a modified version of this ADG, we introduce the concept of a reverse agent dependency in Section 3. This will allow an alternative ordering of Automated Guided Vehicles, while maintaining a collision-free schedule. In Section 4, we formulate the choice of selecting between forward or reverse ADG dependencies as a mixed-integer linear programming problem. The optimization problem formulation guarantees that the resulting ADG allows plan execution to be both collision- and deadlock-free, while minimizing the predicted plan completion time. Finally, we compare this approach to the baseline ADG method in Section 5.

2 Preliminaries

Let us now introduce the fundamental concepts on which our approach is based, facilitated by the example shown in Fig. 2. Consider the representation of a workspace by a roadmap $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ a set of edges, e.g., as in Fig. 2a.

***Definition* 1** (MAPF Solution)**.** *The roadmap $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is occupied by a set of N AGVs where the i^{th} AGV has start $s_i \in \mathcal{V}$ and goal $g_i \in \mathcal{V}$, such that $s_i \neq s_j$ and $g_i \neq g_j \ \forall \ i, j \in \{1, \dots, N\}, \ i \neq j$. A MAPF solution $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_N\}$ is a set of N plans, each defined by a sequence $\mathcal{P} = \{p^1, \dots, p^{N_i}\}$ of tuples $p = (l, t)$, with a location $l \in \mathcal{V}$ and a time $t \in [0, \infty)$. The MAPF solution is such that, if every AGV perfectly follows its plan, then all AGVs will reach their respective goals in finite time without collision.*

For a plan tuple $p = (l, t)$, let us define the operators $l = loc(p)$ and $t = \hat{t}(p)$ which return the location $l \in \mathcal{V}$ and *planned* time of plan tuple p respectively. Let $S(l) \mapsto S \subset \mathbb{R}^2$ be an operator which maps a location l (obtained from $l = loc(p)$) to a spatial region in the physical workspace in $\mathbb{R}^2$. Let $S_{\text{AGV}} \subset \mathbb{R}^2$ refer to the physical area occupied by an AGV.

In Fig. 2a, AGV_1 and AGV_2 have start and goal $s_1 = A$, $g_1 = H$ and $s_2 = E$, $g_2 = D$, respectively. For this example, using CCBS (Andreychuk et al. 2019) yields $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2\}$ as

$$\mathcal{P}_1 = \{(A, 0), (B, 1.0), (C, 2.2), (G, 3.1), (H, 3.9)\},$$
$$\mathcal{P}_2 = \{(E, 0), (F, 1.1), (G, 3.9), (C, 4.8), (D, 5.9)\}.$$

Note the implicit ordering in $\mathcal{P}$, stating AGV_1 traverses $C - G$ before AGV_2.

2.1 Modified Action Dependency Graph

Based on a MAPF solution $\mathcal{P}$, we can construct a modified version of the original Action Dependency Graph (ADG), formally defined in Definition 2. This modified ADG encodes the sequencing of AGV movements to ensure the plans are executed as originally planned despite delays.

***Definition* 2** (Action Dependency Graph). *An ADG is a directed graph $\mathcal{G}_{ADG} = (\mathcal{V}_{ADG}, \mathcal{E}_{ADG})$ where the vertices represent events of an AGV traversing a roadmap $\mathcal{G}$. A vertex $v_i^k = (\{p_1, \ldots, p_q\}, \text{status}) \in \mathcal{V}_{ADG}$ denotes the k^{th} event of the i^{th} AGV moving from $loc(p_1)$, via intermediate locations, to $loc(p_q)$, where $q \geq 2$ denotes the number of consecutive plan tuples encoded within v_i^k.* status $\in$ {staged, in-progress, completed}. *The edge $(v_i^k, v_j^l) \in \mathcal{E}_{ADG}$, from here on referred to as a dependency, states that v_j^l cannot be* in-progress *or* completed *until $v_i^k =$* completed. *An edge $(v_i^k, v_j^l) \in \mathcal{E}_{ADG}$ is classified as* Type 1 *if $i = j$ and* Type 2 *if $i \neq j$.*

Initially, the *status* of v_i^k are *staged* $\forall\ i, k$. Let us introduce $plan(v_i^k)$ which returns the sequence of plan tuples $\{p_1, \ldots, p_q\}$ for $v_i^k \in \mathcal{V}_{\text{ADG}}$. Let the operators $s(v_i^k)$ and $g(v_i^k)$ return the start and goal vertices $loc(p_1)$ and $loc(p_q)$ of vertex v_i^k respectively and $\oplus$ denote the Minkowski sum. We also differentiate between *planned* and *actual* ADG vertex completion times. Let $\hat{t}_s(v_i^k)$ and $\hat{t}_g(v_i^k)$ denote the *planned* time that event $v_i^k \in \mathcal{V}_{\text{ADG}}$ starts (*status* changes from *staged* to *in-progress*) and is completed (*status* changes from *in-progress* to *completed*), respectively. An ADG can be constructed from a plan $\mathcal{P}$ using Algorithm 1. AGVs can execute their plans as originally described by the MAPF solution $\mathcal{P}$ by adhering to the ADG, defined next in Definition 3.

***Definition* 3** (Executing ADG based plans). *AGVs adhere to the ADG if each AGV only starts executing an ADG event v_i^k (status of v_i^k changes from* staged *to* in-progress*) if all dependencies pointing to v_i^k have* status = completed *for all $v_i^k \in \mathcal{V}_{ADG}$.*

Fig. 2b shows an example of AGVs adhering to the ADG. Observe how AGV_2 cannot start v_2^2 before v_1^4 has been completed by AGV_1, as dictated by Definition 3.

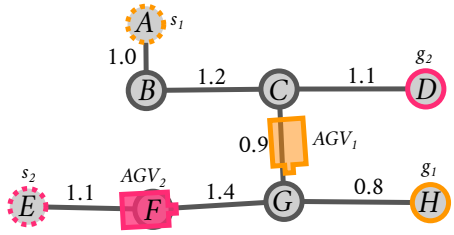
Next, we introduce Assumption 1 which we require to maintain deadlock-free behavior between AGVs when executing an ADG based plan as described in Definition 3.

***Assumption* 1** (Acyclic ADG). *The ADG constructed by Algorithm 1 using $\mathcal{P}$ as defined in Definition 1 is acyclic.*

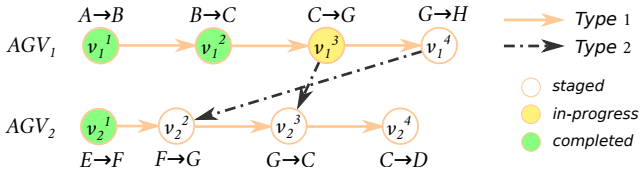
***Remark* 1.** *Assumption 1 can in practice always be satisfied in case the roadmap vertices outnumber the AGV fleet size, i.e. $|\mathcal{V}| > N$ (as is typically the case in warehouse robotics). Simple modificationa to existing MAPF solvers (e.g. an extra edge constraint in CBS) is sufficient to obtain ADGs that satisfy Assumption 1 (Hönig et al. 2019).*

Unlike the originally proposed ADG algorithm, Algorithm 1 ensures that non-spatially-exlusive subsequent plan tuples are contained within a single ADG vertex, cf. line 7 of the algorithm. This property will prove to be useful with the introduction of reverse dependencies in Section 3.1. Despite these modifications, Algorithm 1 maintains the original algorithm's time complexity of $\mathcal{O}(N^2\bar{n}^2)$ where $\bar{n} = \max_i N_i$.

Due to delays, the *planned* and *actual* ADG vertex times may differ. Much like the previously introduced *planned*



(a) A roadmap graph occupied by two AGVs with start s_i and goal g_i for $i = \{1, 2\}$. The start and goal vertices are highlighted with dotted and solid colored circle outlines respectively. The edge weights indicate the expected traversal times.



(b) Illustration of the ADG where each vertex status is color coded. It reflects the momentary progress of the AGVs in Fig. 2a.

Figure 2: Illustrative MAPF problem example alongside the constructed Action Dependency Graph

Algorithm 1 Modified ADG construction based on (Hönig et al. 2019)

Input: MAPF solution $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Result: $\mathcal{G}_{\text{ADG}}$

// Add ADG vertices and *Type 1* dependencies
1: **for** $i = 1$ to N **do**
2: $p \leftarrow p_i^1$
3: $v \leftarrow (\{p\}, staged)$
4: $v_{\text{prev}} \leftarrow None$
5: **for** $k = 2$ to N_i **do**
6: Append p_i^k to $plan(v)$
7: **if** $S(loc(p)) \oplus S_{\text{AGV}} \cap S(loc(p_i^k)) \oplus S_{\text{AGV}} = \emptyset$ **then**
8: Add v to $\mathcal{V}_{\text{ADG}}$
9: **if** v_{prev} not *None* **then**
10: Add edge (v_{prev}, v) to $\mathcal{E}_{\text{ADG}}$
11: $v_{\text{prev}} \leftarrow v$
12: $p \leftarrow p_i^k$
13: $v \leftarrow (\{p\}, staged)$

// Add *Type 2* dependencies
14: **for** $i = 1$ to N **do**
15: **for** $k = 1$ to N_i **do**
16: **for** $j = 1$ to N **do**
17: **if** $i \neq j$ **then**
18: **for** $l = 1$ to N_j **do**
19: **if** $s(v_i^k) = g(v_j^l)$ **and** $\hat{t}_g(v_i^k) \leq \hat{t}_g(v_j^l)$ **then**
20: Add edge (v_i^k, v_j^l) to $\mathcal{E}_{\text{ADG}}$
21: **return** $\mathcal{G}_{\text{ADG}}$

event start and completion times $\hat{t}_s(v_i^k)$ and $\hat{t}_g(v_i^k)$, we also introduce $t_s(v_i^k)$ and $t_g(v_i^k)$ which denote the *actual* start and completion times of event $v_i^k \in \mathcal{V}_{\text{ADG}}$ respectively. Note that if the MAPF solution is executed nominally, i.e. AGVs experience no delays, then $t_s(v_i^k) = \hat{t}_s(v_i^k)$ and $t_g(v_i^k) = \hat{t}_g(v_i^k)$ for all $v_i^k \in \mathcal{V}_{\text{ADG}}$. Let us introduce an important property of an ADG-managed plan-execution scheme, Proposition 1, concerning guarantees of successful plan execution.

***Proposition* 1** (Collision- and deadlock-free ADG plan execution)**.** *Consider an ADG, $\mathcal{G}_{ADG}$, constructed from a MAPF solution as defined in Definition 1 using Algorithm 1, satisfying Assumption 1. If the AGV plan execution adheres to the dependencies in $\mathcal{G}_{ADG}$, then, assuming the AGVs are subjected to a finite number of delays of finite duration, the plan execution will be collision-free and completed in finite time.*

Proof 1*: Proof by induction. Consider that AGV_i and AGV_j traverse a common vertex $\bar{p} \in \mathcal{G}$ along their plans $\mathcal{P}_i$ and $\mathcal{P}_j$, for any $i, j \in \{1, \dots, N\}, i \neq j$. By lines 1-13 of Algorithm 1, this implies $g(v_i^k) = s(v_j^l) = \bar{p}$ for some $v_i^k, v_j^l \in \mathcal{V}_{ADG}$. By lines 14-20 of Algorithm 1, common vertices of $\mathcal{P}_i$ and $\mathcal{P}_j$ in $\mathcal{G}$ will result in a Type 2 dependency (v_j^l, v_i^k) if $p = s(v_j^l) = g(v_i^k)$ and $\hat{t}_g(v_i^k) \leq \hat{t}_g(v_j^l)$. For the base step: initially, all ADG dependencies have been adhered to since v_i^1 is staged $\forall\ i \in \{1, \dots, N\}$. For the inductive step: assuming vertices up until v_i^{k-1} and v_j^{l-1} have been completed in accordance with all ADG dependencies, it is sufficient to ensure AGV_i and AGV_j will not collide at $\bar{p}$ while completing v_i^k and v_j^l respectively, by ensuring $t_s(v_i^k) > t_g(v_j^l)$. By line 19 of Algorithm 1 the Type 2 dependency (v_i^k, v_j^l) guarantees $t_s(v_i^k) > t_g(v_j^l)$. Since, by Assumption 1, the ADG is acyclic, at least one vertex of the ADG can be in-progress at all times. By the finite nominal execution time of the MAPF solution in Definition 1, despite a finite number of delays of finite duration, finite-time plan completion is established. This completes the proof.* □

3 Switching Dependencies in the Action Dependency Graph

We now introduce the concept of a reversed ADG dependency. In the ADG, *Type 2* dependencies essentially encode an ordering constraint for AGVs visiting a vertex in $\mathcal{G}$. The idea is to switch this ordering to minimize the effect an unforeseen delay has on the task completion time of each AGV.

3.1 Reverse *Type 2* Dependencies

We introduce the notion of a reverse *Type 2* dependency in Definition 4. It states that a dependency and its reverse encode the same collision avoidance constraints, but with a reversed AGV ordering. Lemma 1 can be used to obtain a dependency which conforms to Definition 4. Lemma 1 is illustrated graphically in Fig. 3.

***Definition* 4** (Reverse *Type 2* dependency)**.** *Consider a Type 2 dependency $d = (v_i^k, v_j^l)$. d requires $t_s(v_j^l) \geq t_g(v_i^k)$. A reverse dependency of d is a dependency d' that ensures $t_s(v_i^k) \geq t_g(v_j^l)$.*

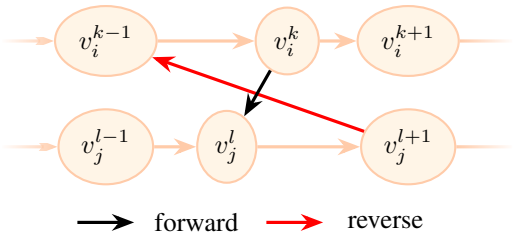


Figure 3: A subset of an ADG with a dependency (black) and its reverse (red)

***Lemma* 1** (Reversed Type 2 dependency)**.** *Let $v_i^k, v_j^l, v_j^{l+1}, v_i^{k-1} \in \mathcal{V}_{ADG}$. Then $d' = (v_j^{l+1}, v_i^{k-1})$ is the reverse dependency of $d = (v_i^k, v_j^l)$.*

Proof 2*: The dependency $d = (v_i^k, v_j^l)$ encodes the constraint $t_s(v_j^l) \geq t_g(v_i^k)$. The reverse of d is denoted as $d' = (v_j^{l+1}, v_i^{k-1})$. d' encodes the constraint $t_s(v_i^{k-1}) \geq t_g(v_j^{l+1})$. By definition, $t_s(v_i^k) \geq t_g(v_i^{k-1})$ and $t_s(v_j^{l+1}) \geq t_g(v_j^l)$. Since $t_g(v) \geq t_s(v)$, this implies that d' encodes the constraint $t_s(v_i^k) \geq t_g(v_j^l)$, satisfying Definition 4.* □

The modified ADG ensures that reverse dependencies maintain collision avoidance since adjacent vertices in $\mathcal{V}_{\text{ADG}}$ refer to spatially different locations, cf. line 7 in Algorithm 1.

3.2 Switchable Action Dependency Graph

Having introduced reverse *Type 2* dependencies, it is necessary to formalize the manner in which we can select dependencies to obtain a resultant ADG. A cyclic ADG implies that two events are mutually dependent, in turn implying a deadlock. To ensure deadlock-free plan execution, it is sufficient to ensure that the selected dependencies result in an acyclic ADG. Additionally, to maintain the collision-avoidance guarantees implied by the original ADG, it is sufficient to select at least one of the forward or reverse dependencies of each forward-reverse dependency pair in the resultant ADG. Since selecting both a forward and reverse dependency always results in a cycle within the ADG, we therefore must either select between the forward or the reverse dependency. To this end, we formally define a Switchable Action Dependency Graph (SADG) in Definition 5 which can be used to obtain the resultant ADG given a selection of forward or reverse dependencies.

***Definition* 5** (Switchable Action Dependency Graph)**.** *Let an ADG as in Definition 2 contain m_T forward-reverse dependency pairs determined using Definition 4. From this ADG we can construct a Switchable Action Dependency Graph SADG($\boldsymbol{b}$) : $\{0,1\}^{m_T} \rightarrow \mathbb{G}$ where $\mathbb{G}$ is the set of all possible ADG graphs obtained by the boolean vector $\boldsymbol{b} = \{b_1, \dots, b_{m_T}\}$, where $b_m = 0$ and $b_m = 1$ imply selecting the forward and reverse dependency of pair m respectively, for $m \in \{1, \dots, m_T\}$.*

***Corollary* 1** (SADG plan execution)**.** *Consider an SADG, $SADG(\boldsymbol{b})$, as in Definition 5. If $\boldsymbol{b}$ is chosen such that $\mathcal{G}_{ADG} = SADG(\boldsymbol{b})$ is acyclic, and no dependencies in $\mathcal{G}_{ADG}$ point from vertices that are staged or in-progress to vertices that are completed, $\mathcal{G}_{ADG}$ will guarantee collision- and deadlock-free plan execution.*

Proof 3*: By definition, any $\boldsymbol{b}$ will guarantee collision-free plans, since at least one dependency of each forward-reverse dependency pair is selected, by Proposition 1. If $\boldsymbol{b}$ ensures $ADG = SADG(\boldsymbol{b})$ is acyclic, and the resultant ADG has no dependencies pointing from vertices that are staged or in-progress to vertices that are completed, the dependencies within the ADG are not mutually constraining, guaranteeing deadlock-free plan execution.*

The challenge is finding $\boldsymbol{b}$ which ensures $SADG(\boldsymbol{b})$ is acyclic, while simultaneously minimizing the cumulative route completion times of the AGV fleet. This is formulated as an optimization problem in Section 4.

4 Optimization-Based Approach

Having introduced the SADG, we now formulate an optimization problem which can be used to determine $\boldsymbol{b}$ such that the resultant ADG is acyclic, while minimizing cumulative AGV route completion times. The result is a Mixed-Integer Linear Program (MILP) which we solve in a closed-loop feedback scheme, since the optimization problem updates the AGV ordering at each iteration based on the delays measured at that time-step.

4.1 Translating a Switchable Action Dependency Graph to Temporal Constraints

Regular ADG Constraints Let us introduce the optimization variable $t^k_{i,s}$ which, once a solution to the optimization problem is determined, will be equal to $t_s(v^k_i)$. The same relation applies to the optimization variable $t^k_{i,g}$ and $t_g(v^k_i)$. The event-based constraints within the SADG can be used in conjunction with a predicted duration of each event to determine when each AGV is expected to complete its plan. Let $\tau(v^k_i)$ be the modeled time it will take AGV_i to complete event $v^k_i \in \mathcal{V}_{\text{ADG}}$ based solely on dynamical constraints, route distance and assuming the AGV is not blocked. For example, we could let τ equal the roadmap edge length divided by the expected nominal AGV velocity. We can now specify the temporal constraints corresponding to the *Type 1* dependencies of the plan of AGV_i as

$$\begin{aligned} t^1_{i,g} &\geq t^1_{i,s} + \tau(v^1_i),\\ t^2_{i,s} &\geq t^1_{i,g},\\ t^2_{i,g} &\geq t^2_{i,s} + \tau(v^2_i),\\ t^3_{i,s} &\geq t^2_{i,g},\\ &\vdots \qquad\qquad \vdots\\ t^{N_i}_{i,s} &\geq t^{N_i-1}_{i,g},\\ t^{N_i}_{i,g} &\geq t^{N_i}_{i,s} + \tau(v^{N_i}_i). \end{aligned} \tag{1}$$

Consider a *Type 2* dependency (v^k_i, v^l_j) within the ADG. This can be represented by the temporal constraint

$$t^l_{j,s} > t^k_{i,g}, \tag{2}$$

where the strict inequality is required to guarantee that AGV_i and AGV_j never occupy the same spatial region.

Adding Switchable Dependency Constraints We now introduce the temporal constraints which represent the selection of forward or reverse dependencies in the SADG. Initially, consider the set $\mathcal{E}^{\text{Type 2}}_{\text{ADG}} = \{e \in \mathcal{E}_{\text{ADG}} | e \text{ is Type 2}\}$ which represents the sets of all *Type 2* dependencies. The aim here is to determine a set $\mathcal{E}^{\text{switchable}}_{\text{ADG}} \subset \mathcal{E}^{\text{Type 2}}_{\text{ADG}}$ containing the dependencies which could potentially be switched and form part of the MILP decision space.

Consider $e_{\text{fwd}} = (v_f, v'_f) \in \mathcal{E}^{\text{Type 2}}_{\text{ADG}}$ and its reverse dependency $e_{\text{rev}} = (v_r, v'_r)$. e_{fwd} and e_{rev} are contained within $\mathcal{E}^{\text{switchable}}_{\text{ADG}}$ if the status of v_f, v'_f, v_r, v'_r is *staged*. An illustrative example of the dependencies contained within $\mathcal{E}^{\text{switchable}}_{\text{ADG}}$ is shown in Fig. 4. Having determined $\mathcal{E}^{\text{switchable}}_{\text{ADG}}$, the next step is to include the switched dependencies as temporal constraints. Directly referring to Section 3.2, we assume m_T forward-reverse dependency pairs in $\mathcal{E}^{\text{switchable}}_{\text{ADG}}$, where the Boolean b_m is used to select the forward or reverse dependency of the mth forward-reverse dependency pair, $m \in \{1, \dots, m_T\}$. These temporal constraints can be written as

$$\begin{aligned} t^l_{j,s} &> t^k_{i,g} - b_m M,\\ t^{k-1}_{i,s} &> t^{l+1}_{j,g} - \big(1 - b_m\big) M, \end{aligned} \tag{3}$$

where M is a large, positive constant such that $M > \max_i t^{N_i}_i$. Note that $\max_i t^{N_i}_i$ can be approximated by estimating the maximum anticipated delays experienced by the AGVs. In practice, however, finding such an upper bound on delays is not evident, meaning we choose M to be a conservatively high value.

4.2 Optimization Problem Formulation

We have shown that an SADG is represented by the temporal constraints in Eq. (1) through Eq. (3) for $i \in \{1, \dots, N\}$, $m \in \{1, \dots, m_T\}$. Minimizing the cumulative route completion time of all AGVs is formulated as the following optimization problem

$$\begin{aligned} \min_{\boldsymbol{b},\, \boldsymbol{t_s},\, \boldsymbol{t_g}} \quad & \sum_{i=1}^{N} t^{N_i}_{i,g}\\ \text{s.t. } & Eq.\ (1)\ \forall\ i = \{1, \dots, N\},\\ & Eq.\ (2)\ \forall\ e \in \mathcal{E}^{\text{Type 2}}_{\text{ADG}} \setminus \mathcal{E}^{\text{switchable}}_{\text{ADG}},\\ & Eq.\ (3)\ \forall\ e \in \mathcal{E}^{\text{switchable}}_{\text{ADG}}, \end{aligned} \tag{4}$$

where $\boldsymbol{b} : \{0,1\}^{m_T}$ is a vector containing all the binary variables b_m and the vectors $\boldsymbol{t_s}$ and $\boldsymbol{t_g}$ contain all the variables $t^k_{i,s}$ and $t^k_{i,g}$ respectively $\forall\ k \in \{1, \dots, N_i\}, i \in \{1, \dots, N\}$.

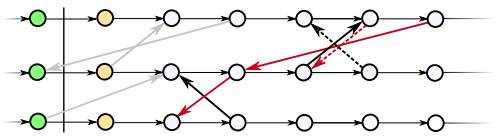


Figure 4: Dependencies contained in $\mathcal{E}_{\text{ADG}}^{\text{switchable}}$, shown in black (forward) and red (reverse). Gray dependencies are in $\mathcal{E}_{\text{ADG}}^{\text{Type 2}} \setminus \mathcal{E}_{\text{ADG}}^{\text{switchable}}$.

4.3 Solving the MILP in a Feedback Loop

The aforementioned optimization formulation can be solved based on the current AGV positions in a feedback loop. The result is a continuously updated $\mathcal{G}_{\text{ADG}}$ which guarantees minimal cumulative route completion times based on current AGV delays. This feedback strategy is defined in Algorithm 2.

An important aspect to optimal feedback control strategies is that of recursive feasibility, which means that the optimization problem will remain feasible as long as the control law is applied. The control strategy outlined in Algorithm 2 is guaranteed to remain recursively feasible, as formally shown in Proposition 2.

Algorithm 2 Switching ADG Feedback Scheme

1: Get goals and locations
2: Solve MAPF to obtain $\mathcal{P}$
3: Construct ADG using Algorithm 1
4: Determine SADG($\boldsymbol{b}$) and set $\boldsymbol{b} = \boldsymbol{0}$ (see Section 4.1)
5: **while** Plans not done **do**
6: get current position along plans for each robot
7: $\boldsymbol{b} \leftarrow$ MILP in Eq. (4)
8: ADG $\leftarrow$ SADG($\boldsymbol{b}$)

***Proposition* 2** (Recursive Feasibility)**.** *Consider an ADG, as defined in Definition 2, which is acyclic at time $t = 0$. Consecutively applying the MILP solution from Eq. (4) is guaranteed to ensure the resultant ADG remains acyclic for all $t > 0$.*

Proof 4*: Proof by induction. Consider an acyclic ADG as defined in Definition 2, at a time t. The MILP in Eq. (4) always has the feasible solution $\boldsymbol{b} = 0$ if the initial ADG (from which the MILP's constraints in Eq. (1) through Eq. (3) are defined) is acyclic. Any improved solution of the MILP with $\boldsymbol{b} \neq 0$ is necessarily feasible, implying a resultant acyclic ADG. This implies that the MILP is guaranteed to return a feasible solution, the resultant ADG will always be acyclic if the ADG before the MILP was solved, was acyclic. Since the ADG at $t = 0$ is acyclic (a direct result of a MAPF solution), it will remain acyclic for $t > 0$.* □

4.4 Decreasing Computational Effort

The time required to solve the MILP will directly affect the real-time applicability of this approach. In general, the complexity of the MILP increases exponentially in the number of binary variables. To render the MILP less computationally demanding, it is therefore most effective to decrease the number of binary variables. We present two complementary methods to achieve this goal.

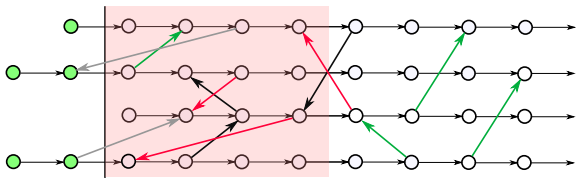


Figure 5: Dependency selection for a horizon of 4 vertices. Switchable dependency pairs are shown in black (forward) and red (reverse). Regular dependencies considered in the MILP are green. Dependencies not considered are gray.

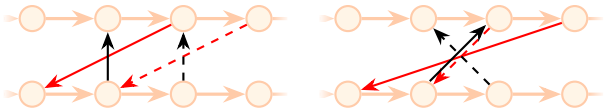
Switching Dependencies in a Receding Horizon Instead of including all switchable dependency pairs in the set $\mathcal{E}_{\text{ADG}}^{\text{switchable}}$, we can only include the switchable dependencies associated with vertices within a horizon H from the last *completed* vertex. An illustration of such selection for $H = 4$ is shown in Fig. 5. Note that dependency selection using this approach maintains ADG acyclicity as in the infinite horizon case, because the set $\mathcal{E}_{\text{ADG}}^{\text{switchable}}$ is smaller, but Eq. (4) remains recursively feasible since the trivial solution guarantees a acyclic ADG at every time-step. Proposition 2 is equally valid when only considering switchable dependencies in a receding horizon. The horizon length H can be seen as a tuning parameter which can offer a trade-off between computational complexity and solution optimality.

Note that, to guarantee recursive feasibility, any switchable dependencies which are not within the horizon H (e.g. the green dependencies in Fig. 5) still need to be considered within the MILP by applying the constraint in Eq. (2). Future work will look into a receding horizon approach that does not necessarily require the consideration of all these constraints while guaranteeing recursive feasibility.

Dependency Grouping We observed that multiple dependencies would often form patterns, two of which are shown in Fig. 6. These patterns are referred to as *same-direction* and *opposite-direction* dependency groups, shown in Fig. 6a and Fig. 6b respectively. These groups share the same property that the resultant ADG is acyclic if and only if either all the forward or all the reverse dependencies are active. This means that a single binary variable is sufficient to describe the switching of all the dependencies within the group, decreasing the variable space of the MILP in Eq. (4). Once such a dependency group has been identified, the temporal constraints can then be defined as

$$\begin{aligned} t_{j,s}^{l} &> t_{i,g}^{k} - b_{DG} M && \forall\, (v_i^k, v_j^l) \in \mathcal{DG}_{\text{fwd}}, \\ t_{j,s}^{l} &> t_{i,g}^{k} - (1 - b_{DG}) M && \forall\, (v_i^k, v_j^l) \in \mathcal{DG}_{\text{rev}}, \end{aligned} \tag{5}$$

where $\mathcal{DG}_{\text{fwd}}$ and $\mathcal{DG}_{\text{rev}}$ refer to the forward and reverse dependencies of a particular grouping respectively, and b_{DG} is a binary variable which switches all the forward or reverse dependencies in the entire group simultaneously.



(a) *same* (b) *opposite*

Figure 6: Dependency groups. Each dependency is either original (black) or reversed (red). Reverse and forward dependency pairings are differentiated by line styles.

5 Evaluation

We design a set of simulations to evaluate the approach in terms of re-ordering efficiency when AGVs are subjected to delays while following their initially planned paths. We use the method presented by Hönig *et al.* (Hönig et al. 2019) as a comparison baseline. All simulations were conducted on a Lenovo Thinkstation with an Intel® Xeon E5-1620 3.5GHz processor and 64 GB of RAM.

5.1 Simulation Setup

The simulations consider a roadmap as shown in Fig. 1. A team of AGVs of size $\{30, 40, 50, 60, 70\}$ are each initialized with a random start and goal position. ECBS (Barer et al. 2014) is used to solve the MAPF with sub-optimality factor $w = 1.6$. We consider delays of duration $k = \{1, 3, 5, 10, 15, 20, 25\}$ time-steps. At the k^{th} time-step, a random subset (20%) of the AGVs are stopped for a length of k. Eq. (4) is solved at each time-step with $M = 10^4$. We evaluate our approach using a Monte Carlo method: for each AGV team size and delay duration configuration, we consider 100 different randomly selected goal/start positions. The receding horizon dependency selection and dependency groups are used as described in Section 4.4.

5.2 Performance Metric and Comparison

Performance is measured by considering the cumulative plan completion time of all the AGVs. This is compared to the same metric using the original ADG approach with no switching as in (Hönig et al. 2019), which is equivalent to forcing the solution of Eq. (4) to $\boldsymbol{b} = \mathbf{0}$ at every time-step. The *improvement* is defined as

$$\text{improvement} = \frac{\sum t_{\text{baseline}} - \sum t_{\text{switching}}}{\sum t_{\text{baseline}}} \cdot 100\%,$$

where $\sum t_*$ refers to the cumulative plan completion time for all AGVs. Note that we consider cumulative plan completion time instead of the make-span because we want to ensure each AGV completes its plans as soon as possible, such that it can be assigned a new task.

Another important consideration is the time it takes to solve the MILP in Eq. (4) at each time-step. For our simulations, the MILP was solved using the academically orientated Coin-Or Branch-and-Cut (CBC) solver (Forrest et al. 2018). However, based on preliminary tests, we did note better performance using the commercial solver Gurobi (Gurobi Optimization, LLC 2020). This yielded computational time improvements by a factor 1.1 up to 20.

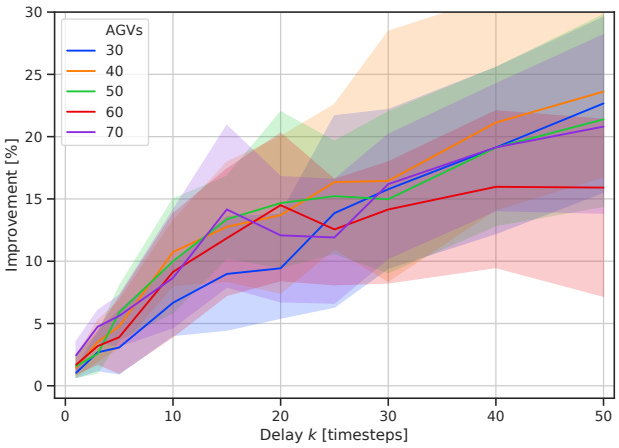


Figure 7: Average improvement of 100 scenarios for various delay lengths and AGV group sizes. Each scenario refers to different randomly generated starts/goals and a randomly selected subset of delayed AGVs. Solid lines depict the average, lighter regions encapsulate the min-max values.

5.3 Results and Discussion

To showcase the efficacy of our approach, we first determine the average improvement of 100 random scenarios using the minimum switching horizon length of 1 for different AGV team sizes and delay lengths, shown in Fig. 7. The average improvement is highly correlated to the delay duration experienced by the AGVs.

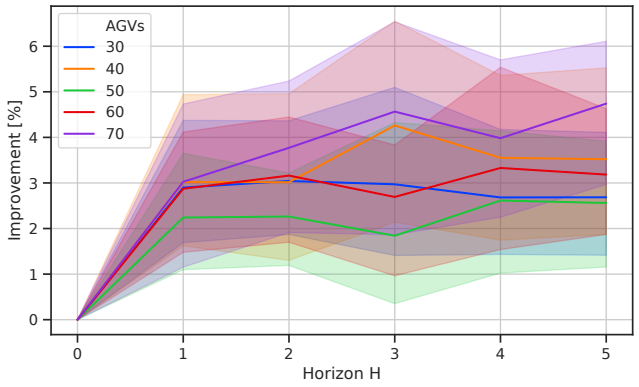
Considering Fig. 7, it is worth noting how the graph layout and AGV-to-roadmap density affects the results: the AGV group size of 40 shows the best average improvement for a given delay duration. This leads the authors to believe there is an optimal AGV group size for a given roadmap, which ensures the workspace is both:

1. Not too congested to make switching of dependencies impossible due to the high density of AGVs occupying the map.
2. Not too sparse such that switching is never needed since AGVs are distant from each other, meaning that switching rarely improves task completion time.

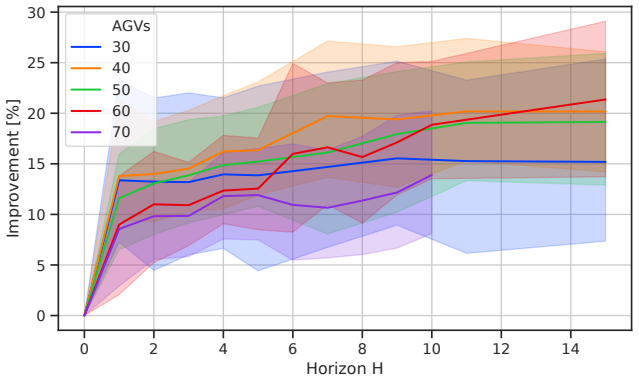
Considering the switching dependency horizon, Fig. 8 shows the average improvement for 100 random start/goal positions and delayed AGV subset selection. We observe that a horizon length of 1 already significantly improves performance, and larger horizons seem to gradually increase performance for larger AGV teams.

Fig. 9 shows the peak computation time for various horizon lengths and AGV team sizes. As expected, the computation time is exponential with horizon size and AGV team size. Two additional observations that were made:

1. *High variability in results.* Note the high variability in improvement indicated by the large lighter regions in Fig. 7. This means that for different random start/goal and delay configurations, the improvement varied significantly. This is due to the fact that each start/goal combination pro-



(a) Delay $k = 3$.



(b) Delay $k = 25$.

Figure 8: Average improvement of 100 random start/goal positions and delayed AGV subset, for different switching horizon lengths, for different AGV group sizes. Solid lines depict the average, lighter regions encapsulate the min-max values.

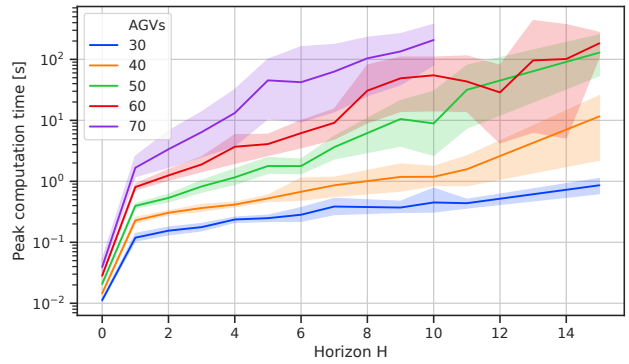


Figure 9: The peak computation to solve the optimization problem for different AGV team sizes and considered dependency horizon lengths. This plot considers the average improvement for delays $k = \{1, 3, 5, 10, 15, 20, 25\}$. Solid lines depict the average, lighter regions encapsulate the min-max values.

vides differing degrees-of-freedom from an ADG switching perspective.

2. *Occasional worse performance.* Occasionally, albeit rarely, our approach would yield a negative improvement for a particular random start/goal configuration. This was typically observed for small delay durations. The reason is that the optimization problem solves the switching assuming no future delays. However, it may so happen that the AGV which was allowed ahead of another, is delayed in the near future, additionally delaying the AGV it surpassed. We believe a robust optimization approach could potentially resolve this.

Finally, we emphasize that our proposed approach:

1. Is a complementary approach which could be used together with the methods presented in (Atzmon et al. 2020; Hönig et al. 2019) and other works.
2. Applies to directional and weighted roadmaps (since ADG switching retains the direction of the original MAPF plan);
3. Persistent planning schemes as in (Hönig et al. 2019), as long as all AGV plans $\mathcal{P}$ are known when the ADG is constructed.

6 Conclusions and Future Work

In this paper, we introduced a novel method which, given a MAPF solution, can be used to switch the ordering of AGVs in an online fashion based on currently measured AGV delays. This switching was formulated as an optimization problem as part of a feedback control scheme, while maintaining the deadlock- and collision-free guarantees of the original MAPF plan. Results show that our approach clearly improves the cumulative task completion time of the AGVs when a subset of AGVs are subjected to delays.

In future work, we plan to consider a receding horizon optimization approach. In this work, ADG dependencies can be switched in a receding horizon fashion, but the plans still need to be of finite length for the optimization problem to be formulated. For truly persistent plans (theoretically infinite length plans), it is necessary to come up with a receding horizon optimization formulation to apply the method proposed in this paper.

Another possible extension is complementing our approach with a local re-planning method. This is because our approach maintains the originally planned trajectories of the AGVs. However, we observed that under large delays, the originally planned routes can become largely inefficient due to the fact the AGVs are in entirely different locations along their planned path. This could potentially be addressed by introducing local re-planning of trajectories.

To avoid the occasional worse performance, we suggest a robust optimization approach to avoid switching dependencies which could have a negative impact on the plan execution given expected future delays.

Finally, to further validate this approach, it is desirable to move towards system-level tests on a real-world intralogistics setup.

References

Andreychuk, A.; Yakovlev, K.; Atzmon, D.; and Stern, R. 2019. Multi-agent pathfinding with continuous time. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 39–45. International Joint Conferences on Artificial Intelligence Organization.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020. Robust Multi-Agent Path Finding and Executing. *Journal of Artificial Intelligence Research* 67:549–579.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 961–962.

Bogatarkan, A.; Patoglu, V.; and Erdem, E. 2019. A declarative method for dynamic multi-agent path finding. In Calvanese, D., and Iocchi, L., eds., *GCAI 2019. Proceedings of the 5th Global Conference on Artificial Intelligence*, volume 65 of *EPiC Series in Computing*, 54–67. EasyChair.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N. R.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA*, 29–37. AAAI Press.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, S.; and Koenig, S. 2018. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 83–87.

Forrest, J.; Ralphs, T.; Vigerske, S.; LouHafer; Kristjansson, B.; jpfasano; EdwinStraver; Lubin, M.; Santos, H. G.; rlougee; and Saltzman, M. 2018. coin-or/cbc: Version 2.9.9.

Gurobi Optimization, LLC. 2020. Gurobi optimizer reference manual.

Hönig, W.; Kumar, S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2017. Summary: Multi-agent path finding with kinematic constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 4869–4873.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J.; and Ayanian, N. 2019. Persistent and robust execution of MAPF schedules in warehouses. *IEEE Robotics and Automation Letters* 4(2):1125–1131.

Lam, E.; Bodic, P. L.; Harabor, D.; and Stuckey, P. 2019. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, (in print).

Li, J.; Harabor, D.; Stuckey, P.; Ma, H.; and Koenig, S. 2019. Symmetry-breaking constraints for grid-based multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, (in print).

Ma, H.; Kumar, S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 3605–3612.

Morris, R.; Pasareanu, C. S.; Luckow, K. S.; Malik, W.; Ma, H.; Kumar, T. K. S.; and Koenig, S. 2016. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop: Planning for Hybrid Systems*.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games* 5(4):293–311.

Pecora, F.; Andreasson, H.; Mansouri, M.; and Petkov, V. 2018. A loosely-coupled approach for multi-robot coordination, motion planning and control. In *Twenty-eighth international conference on automated planning and scheduling*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219:40 – 66.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. *CoRR* abs/1906.08291.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. volume 29 of *AI Magazine*, 9 – 19. Menlo Park, CA: AAAI.

Yakovlev, K. S., and Andreychuk, A. 2017. Any-angle pathfinding for multiple agents based on SIPP algorithm. *CoRR* abs/1703.04159.

Yu, J., and LaValle, S. M. 2012. Multi-agent Path Planning and Network Flow. *CoRR* abs/1204.5717.

Yu, J., and LaValle, S. M. 2013. Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation*, 3612–3617.

Token-based Execution Semantics for Multi-Agent Epistemic Planning

Thorsten Engesser , **Robert Mattmüller** , **Bernhard Nebel** , **Felicitas Ritter**
University of Freiburg, Germany
{engesser,mattmuel,nebel,ritterf}@cs.uni-freiburg.de

Abstract

Epistemic planning has been employed as a means to achieve implicit coordination in cooperative multi-agent systems where world knowledge is distributed between the agents, and agents plan and act individually. However, recent work has shown that even if all agents act with respect to plans that they consider optimal from their own subjective perspective, infinite executions can occur. In this paper, we analyze the idea of using a single token that can be passed around between the agents and which is used as prerequisite for acting. We show that introducing such a token to any planning task will prevent the existence of infinite executions. We furthermore analyze the conditions under which solutions to a planning task are preserved under our tokenization.

1 Introduction

Epistemic implicit coordination planning (Engesser et al. 2017) is a technique for planning and coordination in multi-agent systems in which agents try to collaboratively reach a joint goal. The knowledge and abilities required to reach the goal can be distributed among the agents. With no centralized coordination instance and without the possibility for the agents to agree on a joint plan, the agents need to plan individually and execute their plans in a decentralized way.

A central assumption Engesser et al. (2017) make for their notion of policies and policy execution is that actions are applied in sequence by the agents. However, the order in which agents are allowed to act is not preimposed, i.e., situations where multiple agents have an applicable action are allowed.

The advantage of this kind of sequentiality over joint actions is that for a problem to be solvable, solution existence does not have to be common knowledge between all agents. Instead, only the agent that performs the first action of a plan has to know that the plan leads to the goal and that after the execution of the first action, the next agent who is designated to act will know, and so on. This is helpful since often it is not known in advance which orders of agents will work. In such cases, any agent that finds a plan can begin, with the other agents waiting until they have sufficient information.

One issue with this approach is that agents who each plan for themselves and act according to their own policies may want to act at the same time. These kinds of "conflicts of interest" are not considered at the policy level but at the execution level. Since the idea of planning for implicit coordination is that there should be no centralized instance coordinating the agents, one has to consider all possible executions resulting from each order in which the agents could act, given each of their individual plans. While this *interleaving* semantics is harmless in some cases, in other cases it results in agents inadvertently working against each other.

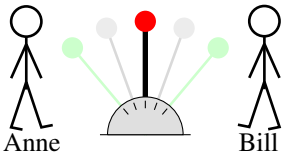


Figure 1: Two agents and a lever.

Consider, for example, the situation depicted in Figure 1, in which a lever can be pulled left or right, with both the leftmost and the rightmost position being a goal state. If Anne's plan is to pull the lever all the way to the left and Bill's plan is to pull the lever all the way to the right, there are executions in which the lever is pulled back and forth indefinitely. An obvious fix that works in this particular instance is to require the agents to act only with respect to optimal plans and thus pull the lever only towards the nearest goal configuration. However, Bolander et al. (2018) have shown that problems with infinite executions can still occur if the agents' knowledge about the world differs. E.g., we could have the situation in which both the leftmost and the rightmost configuration can be, but are not necessarily, goal configurations. Imagine Anne only knows about whether or not the leftmost configuration is a goal configuration and Bill only knows about whether or not the rightmost configuration is one. Using our solution concept, both have to assume the worst case of their configuration being the only goal configuration and we still end up with infinite executions.

Instead of trying to provide success guarantees by only restricting the types of policies which the agents are allowed to take, in the present paper, we try to tackle the problem of infinite executions on the planning task level. The reason why requiring policies to be optimal does not always prevent infinite executions is that optimality is judged from the subjective perspective of each agent. Thus an agent may act because from his perspective the action is part of an optimal

plan, while from the perspective of the agent who acted before it is not, and another agent was expected to act instead. To prevent this from happening, we exclude all agents except for one from performing actions. The acting agent can then specify the agent who is allowed to act next. To model this, we introduce a token that can be passed around by the agents. Only the agent with the token is allowed to perform an action. We show that this approach does not only solve the lever problem from the example but prevents infinite execution in general, while preserving plan existence, given some formal criteria are met.

The remainder of this paper is structured as follows: First, we are going to give a brief overview of related work, and highlight how this paper builds on it. Next, we will introduce the formal framework, dynamic epistemic logic (DEL). In the main part of the paper, we demonstrate how to rewrite a given planning task to include tokens, show that this eliminates infinite executions, and discuss under which conditions plan existence can be preserved. Finally, we conclude and discuss future work.

2 Related Work

In this paper, we attempt to provide success guarantees of interleaved plan executions, just like Bolander et al. (2018) did. Unlike them, we do not study how *agent types* impact the success of implicitly coordinated plans, but rather impose restrictions on allowed behavior by modifying the *rules of the planning task*. Imposing the rule that only an agent that possesses the token may act overcomes a limitation of the agent-types approach: Without tokens, even optimally eager agents are only guaranteed to prevent infinite executions if there is uniform observability.

In recent work orthogonal to the tokenization approach we present here, Nebel et al. (2019) investigated how implicitly coordinated plans without communications can succeed in the *special case* of multi-agent path finding with destination uncertainty, in which agents have to move to different destinations in a collision free manner without communicating, while the agents' individual destinations are *not* common knowledge among them. It was shown that, in such scenarios, eagerness and the capability to perform conservative re-planning, are sufficient to ensure that plans succeed.

Besides planning with the intent to allow agents to self-coordinate, epistemic planning has been mostly applied to finding centralized plans in the presence of knowledge preconditions and goals (Kominis and Geffner 2015; Muise et al. 2015; Huang et al. 2017; Le et al. 2018). In more recent work, Maubert, Pinchinat, and Schwarzentruber (2019) have looked at modeling and synthesizing strategies for reachability games in DEL, which is a setting which is more similar to ours. While in their formalism, agents also act sequentially, there is no interleaving concurrency and it is assumed that it is always commonly known which agent's turn it is.

Tokens have also made an appearance in distributed systems, for example in the work of Loucks and Shaheen (1997). Components of a distributed system are located apart from each other, but still have to communicate and coordinate their actions to achieve a common goal. Makki et al. (1992) used a token queue and semaphore for restricting the use of a mutual resource that can only have a small number of users at a time. In contrast to the approach taken in the present paper, their tokens are used to restrict the access to a limited resource. Their agents do not plan with other agents, and handing the token to the next player is usually done by a waiting queue. In our approach, the idea is that the agent who has the token gets to decide which agent will have the token next. This is not desired in distributed systems because they want all agents to have the same rights of access to a resource with no agent being strategically excluded.

Adding tokens can be seen as implementing a simple and practical *social law* (specifically the law that only allows an agent to act if it possess the token), a concept that has gained increasing interest in multi-agent planning lately. Social laws such as the ones from Karpas, Shleyfman, and Tennenholtz (2017) and Nir and Karpas (2019) have also tried to minimize the problems that arise from multiple agents by trying to force the agents to work together and minimize the amount of "damage" agents can do if they want to prevent other agents from reaching their goal. Unlike their social laws, which have to be designed by a rational person, and specially made to fit one specific planning task, our approach can be applied in a generalized way to given planning tasks.

3 Epistemic Planning

In the following, we will recapitulate the syntax and semantics of Dynamic Epistemic Logic (DEL) (van Ditmarsch, van der Hoek, and Kooi 2007), which we will use as the formal framework of this paper. We will use the conventions of Bolander et al. (2018), and also use their definitions of planning tasks, policies, agent types and executions.

Let $\mathcal{A}$ be a finite set of agents and P be a finite set of atomic propositions. The *epistemic language* $\mathcal{L}_{\text{KC}}$ is then defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i\varphi \mid C\varphi$$

with $p \in P$ and $i \in \mathcal{A}$. Formula $K_i\varphi$ reads as "agent i knows φ" and $C\varphi$ reads as "it is common knowledge that φ". Furthermore, the operators $\top, \bot, \leftarrow, \rightarrow, \leftrightarrow$ are defined as abbreviations, analogously to their definition in propositional logic. We will refrain from specifying P and $\mathcal{A}$ explicitly, if their values are clear from the context.

Example 1. Recall the lever example from the introduction. If we ignore the position of the lever, we can model the situation using $\mathcal{A} = \{anne, bill\}$, and $P = \{l, r\}$, where l denotes whether there is a goal position to the left and r denotes whether there is a goal position to the right. The formula $\neg K_{anne} r \wedge \neg K_{anne} \neg r$ expresses that Anne does not know whether or not there is a goal to the right. The formula $K_{bill}(\neg K_{anne} r \wedge \neg K_{anne} \neg r)$ expresses that Bill does know that Anne does not know.

Epistemic formulas are evaluated in *epistemic models* $\mathcal{M} = \langle W, (\sim_i)_{i\in\mathcal{A}}, V\rangle$, where W is a non-empty finite set of worlds (called the *domain* of $\mathcal{M}$), $\sim_i \subseteq W \times W$ is an equivalence relation called the *indistinguishability relation* for each agent $i \in \mathcal{A}$, and $V : P \to \mathcal{P}(W)$ is the *valuation function*, assigning to each proposition $p \in P$ a set of worlds $V(p)$ in which the proposition is true.

We depict epistemic models as graphs where the nodes correspond to worlds and the edges correspond to indistinguishability between the worlds. Nodes are labeled with the world name and all propositions which are true in that world. Edges are labeled with the agents for which the worlds are indistinguishable. For better readability, we usually omit reflexive edges and edges that are implied by transitivity.

Example 2. Assuming that it is common knowledge that at least one of the two positions must be a goal position, we can model the situation from our running example as epistemic model $\mathcal{M}_0$ with three worlds: one world w_1 in which just the left position is a goal, one world w_3 in which just the right position is a goal and one world w_2 in which both positions are a goal. The epistemic model, including the indistinguishabilities for the agents is depicted below:

$$\mathcal{M}_0 = \quad \underset{w_1 : l}{\bullet} \overset{anne}{\text{———}} \underset{w_2 : l, r}{\bullet} \overset{bill}{\text{———}} \underset{w_3 : r}{\bullet}$$

For $W_d \subseteq W$, the pair $(\mathcal{M}, W_d)$ is called an *epistemic state* (or simply a state) and the worlds of W_d are called *designated worlds*. A state is called *global* if $W_d = \{w\}$ for some world w (called the *actual world*). We then often write $(\mathcal{M}, w)$ instead of $(\mathcal{M}, \{w\})$. We use $S^{gl}(P, \mathcal{A})$ to denote the set of global states (or simply S^{gl} if P and $\mathcal{A}$ are clear from context). For any state $s = (\mathcal{M}, W_d)$ we let $\mathit{Globals}(s) = \{(\mathcal{M}, w) \mid w \in W_d\}$. A state $(\mathcal{M}, W_d)$ is called a local state for agent i if W_d is closed under $\sim_i$ (that is, if $w \in W_d$ and $w \sim_i v$, then $v \in W_d$).
Given a state $s = (\mathcal{M}, W_d)$ the associated local state of agent i, denoted s^i, is $(\mathcal{M}, \{v | v \sim_i w \text{ and } w \in W_d\})$. Going from s to s^i amounts to a *perspective shift* to the local perspective of agent i.

Example 3. Let $s_0 = (\mathcal{M}_0, w_2)$ be the global state for the lever example. Then Anne sees the local state $s_0^{anne} = (\mathcal{M}_0, \{w_1, w_2\})$, meaning she cannot distinguish whether $(\mathcal{M}_0, w_1)$ or $(\mathcal{M}_0, w_2)$ from $\mathit{Globals}(s_0^{anne})$ is the true global state. Bill sees $s_0^{bill} = (\mathcal{M}_0, \{w_2, w_3\})$.

Let $(\mathcal{M}, W_d)$ be a state with $\mathcal{M} = \langle W, (\sim_i)_{i \in \mathcal{A}}, V \rangle$. For $i \in \mathcal{A}$, $p \in P$ and $\varphi, \psi \in \mathcal{L}_{\text{KC}}$, truth is defined as follows:

$$\begin{array}{lll}
(\mathcal{M}, W_d) \models \varphi & \text{iff} & (\mathcal{M}, w) \models \varphi \text{ for all } w \in W_d \\
(\mathcal{M}, w) \models p & \text{iff} & w \in V(p) \\
(\mathcal{M}, w) \models \neg\varphi & \text{iff} & (\mathcal{M}, w) \not\models \varphi \\
(\mathcal{M}, w) \models \varphi \wedge \psi & \text{iff} & (\mathcal{M}, w) \models \varphi \text{ and } (\mathcal{M}, w) \models \psi \\
(\mathcal{M}, w) \models K_i\varphi & \text{iff} & (\mathcal{M}, v) \models \varphi \text{ for all } v \sim_i w \\
(\mathcal{M}, w) \models C\varphi & \text{iff} & (\mathcal{M}, v) \models \varphi \text{ for all } v \sim^* w
\end{array}$$

where $\sim^*$ is the transitive closure of $\bigcup_{i \in \mathcal{A}} \sim_i$.

Example 4. We can now verify for our running example, that it indeed holds that $s_0 \models \neg K_{anne} r \wedge \neg K_{anne} \neg r$. Note that checking a formula $K_i\varphi$ in a state s amounts to the same as checking the formula φ in s^i. In our example, $s_0 \not\models K_{anne} r$ because $(\mathcal{M}_0, \{w_1, w_2\}) \not\models r$, and $s_0 \not\models K_{anne} \neg r$ because $(\mathcal{M}_0, \{w_1, w_2\}) \not\models \neg r$.

Note that syntactically different states can be epistemically equivalent, i.e., satisfy the exact same set of epistemic formulas. In the following, we assume that such states are identified. In practice, one can do that by checking for *bisimilarity* (Blackburn, de Rijke, and Venema 2001).

3.1 Epistemic Actions and the Product Update

We also need a way do define actions, which can change the facts of the world as well as the knowledge of the agents. The way this is done in the action model logic of DEL is using so-called *event models*.

An event model is a 4-tuple $\mathcal{E} = \langle E, (\sim_i)_{i \in \mathcal{A}}, \text{pre}, \text{eff} \rangle$ where E is a non-empty finite set of events (called the *domain* of $\mathcal{E}$), $\sim_{\mathcal{A}} \subseteq E \times E$ is an equivalence relation called the indistinguishability relation for each agent $i \in \mathcal{A}$, and the functions $\text{pre} : E \to \mathcal{L}_{\text{KC}}$ and $\text{eff} : E \to \mathcal{L}_{\text{KC}}$ assign *preconditions* and *effects* to each event. While for each event $e \in E$, the precondition $\text{pre}(e)$ can be an arbitrary formula from $\mathcal{L}_{\text{KC}}$, the effect $\text{eff}(e)$ is a conjunction of literals, i.e, of atomic propositions and their negations, including $\top$ and $\bot$.

Each event of an action represents a different possible outcome. By using multiple events $e, e' \in E$ which are indistinguishable ($e \sim_i e'$), for some agent $i \in \mathcal{A}$, it is possible to model actions that are only partially observable.

We depict event models similarly to epistemic models as graphs, where the nodes correspond to events and the edges correspond to the indistinguishability between events. We label each node for an event $e \in E$ with $e : \langle \text{pre}(e), \text{eff}(e) \rangle$. As before, we usually omit reflexive edges and edges that are implied by transitivity for better readability.

For $E_d \subseteq E$, the pair $(\mathcal{E}, E_d)$ is called an *epistemic action*, or simply action. We call $(\mathcal{E}, E_d)$ a local action for agent i when E_d is closed under $\sim_i$.

Example 5. Consider the following event model which we will use to model a *sensing* action for Anne. It contains one event for each possible sensing outcome. Event e_1 occurs if r is true and event e_2 occurs if r is false. Since the action should not change any facts, both events have the effect $\top$. There is no indistinguishability between e_1 and e_2 for Anne. This way, after the action, she will know whether e_1 or e_2 has occurred and thus whether r is true or false. To make the action as general as possible we leave the events indistinguishable to Bill: If he does not know whether or not r is true, he should not learn it as result of Anne's sensing action.

$$\mathcal{E}_1 = \quad \underset{e_1 : \langle r, \top \rangle}{\bullet} \overset{bill}{\text{———}} \underset{e_2 : \langle \neg r, \top \rangle}{\bullet}$$

Anne's sensing action is then $(\mathcal{E}_1, \{e_1, e_2\})$. We need to designate both worlds due to not knowing the outcome of a sensing action in advance. The action $(\mathcal{E}_1, \{e_1\})$ could also make sense, e.g., as an action for a third agent informing Anne that r is true, without letting Bill know about it.

The semantics of action application is given by the *product update*. Let a state $s = (\mathcal{M}, W_d)$ and an action $a = (\mathcal{E}, E_d)$ be given with $\mathcal{M} = \langle W, (\sim_i)_{i \in \mathcal{A}}, V \rangle$ and $\mathcal{E} = \langle E, (\sim_i)_{i \in \mathcal{A}}, \text{pre}, \text{eff} \rangle$. Then the product update of s with a is defined as $s \otimes a = (\langle W', (\sim'_i)_{i \in \mathcal{A}}, V' \rangle, W'_d)$ where

- each world is paired up with all applicable events, i.e., $W' = \{(w, e) \in W \times E \mid \mathcal{M}, w \models \text{pre}(e)\}$;

- new worlds are indistinguishable if the old worlds were indistinguishable and the events are indistinguishable, i.e., $(w, e) \sim_i' (w', e')$ iff $w \sim_i w'$ and $e \sim_i e'$;
- propositions become true if they occur positively in the effect of the event, or if they don't occur negatively and have already been true before, i.e., $(w, e) \in V'(p)$ iff $\mathrm{eff}(e) \models p$ or $(\mathcal{M}, w \models p$ and $\mathrm{eff}(e) \not\models \neg p)$;
- worlds are designated if both predecessor world and event are designated, i.e., $(w, e) \in W_d'$ iff $w \in W_d$ and $e \in E_d$.

We say that an action $a = (\mathcal{E}, E_d)$ is *applicable* in a state $s = (\mathcal{M}, W_d)$ if for all $w \in W_d$ there is an applicable event $e \in E_d$, meaning $\mathcal{M}, w \models \mathrm{pre}(e)$.

Example 6. In our running example, the action a_1 is applicable in s_0, and after the action application, we obtain the state $s_1 = (\mathcal{M}_1, \{(w_2, e_1)\})$ where $\mathcal{M}_1$ is depicted below:

$$\mathcal{M}_1 = \quad \underset{(w_1,e_2):l}{\bullet} \qquad \underset{(w_2,e_1):l,r}{\bullet} \overset{bill}{\text{———}} \underset{(w_3,e_1):r}{\bullet}$$

We can see that in s_1, Anne now knows that r. Also, while Bill has not learned anything new about l, he knows now that Anne knows about r. The sensing action of Anne is *public* in the sense that other agents will know that the sensing has taken place.

3.2 Planning Tasks, Policies and Executions

We now have everything that is needed to define multi-agent epistemic planning tasks in DEL. For a fixed set of agents $\mathcal{A}$, a *planning task* $\Pi = \langle s_0, A, \omega, \gamma \rangle$ consists of a global state s_0 called the *initial state*; a finite set of actions A; an *owner function* $\omega : A \to \mathcal{A}$; assigning each action to its owner and a *goal formula* $\gamma \in \mathcal{L}_{\mathrm{KC}}$.

Example 7. For the lever problem, the planning task is a tuple $\langle s_0, A, \omega, \gamma \rangle$ such that s_0 is defined as before plus additional propositions that indicate the position of the lever (e.g., $p_l \in P$ to indicate that the lever is at the leftmost position, $p_r \in P$ to indicate that the lever is at the rightmost position, and other propositions for the positions in between). For each non-leftmost position of the lever, we could then have an action $a \in A$ for pulling the lever to the left. These actions are owned by Anne, i.e., $\omega(a) = \mathit{anne}$. And for all non-rightmost positions we could have actions $a \in A$ for Bill pulling the lever to the right, i.e., with $\omega(a) = \mathit{bill}$. Since these pulling actions can be fully observed by all agents, they could be defined using only one single event with an appropriate precondition and effect. Finally, the goal formula would be $\gamma = (l \wedge p_l) \vee (r \wedge p_r)$.

A policy π for $\Pi = \langle s_0, A, \omega, \gamma \rangle$ is a partial mapping $\pi : S^{gl} \hookrightarrow \mathcal{P}(A)$ satisfying the conditions applicability (appl), uniformity (unif), and single-agent determinism (det).

(appl) Actions are applicable in states they are assigned to: for all $s \in S^{gl}, a \in \pi(s) : a$ is applicable in s.

(unif) If in some state the policy prescribes an action to an agent, it should prescribe the action also in states that the agent cannot distinguish: for all $s, t \in S^{gl}$ such that $s^{\omega(a)} = t^{\omega(a)}$, from $a \in \pi(s)$ follows $a \in \pi(t)$.

(det) For each state, the policy assigns at most one action per agent. I.e., there are no $s \in S^{\mathrm{gl}}$ and $a, a' \in \pi(s)$ with $a \neq a'$ and $\omega(a) = \omega(a')$.

Note that our definition of uniformity works because we consider bisimilar states to be equal. For two epistemically equivalent states s and s', it thus holds that $\pi(s) = \pi(s')$. The properties uniformity and applicability together imply *knowledge of preconditions*, the property that in each state, an agent who is supposed to perform a particular action must also know that the action is applicable in that state.

Note also that because of the uniformity we must allow policies to sometimes prescribe multiple actions of different owners to the same state. Imagine that from some state s the goal can be only reached via action a of agent i and from some state s' the goal can be only reached via action b of agent j. If there is a state s'' which is indistinguishable to s for agent i and to s' for agent j, then the policy should assign both actions a and b to state s''. To characterize the different outcomes of agents acting according to a common policy, we define the notion of policy executions:

An execution of a policy π from a global state s_0 is a maximal (finite or infinite) sequence of alternating global states and actions $(s_0, a_1, s_1, a_2, s_2, ...)$, such that for all $m \geq 0$,

(1) $a_{m+1} \in \pi(s_m)$, and

(2) $s_{m+1} \in \mathit{Globals}(s_m \otimes a_{m+1})$.

An execution is called successful for a planning task $\Pi = \langle s_0, A, \omega, \gamma \rangle$, if it is a finite execution $(s_0, a_1, s_1, ..., a_n, s_n)$ such that $s_n \models \gamma$.

Example 8. In the lever example, a policy could be, starting in the position with the lever being in the middle, Bill pulls the lever to the right and then to the right again. This policy satisfies all policy properties and its only execution is successful since the goal formula is satisfied in the end. However, while from the perspective of Bill, this is a reasonable policy, Anne cannot verify that the policy is successful because she does not know whether or not the right position is a goal position.

We now want to restrict our focus to policies that are guaranteed to achieve the goal after a finite number of steps. More formally, all of their executions must be successful. As in nondeterministic planning, such policies are called strong (Cimatti et al. 2003). For a planning task $\Pi = \langle s_0, A, \omega, \gamma \rangle$, a policy π is called strong if $s_0 \in \mathrm{Dom}(\pi) \cup \{s \in S^{gl} \mid s \models \gamma\}$ and for each $s \in \mathrm{Dom}(\pi)$, any execution of π from s is successful for Π. A planning task is called solvable if a strong policy for Π exists. For $i \in \mathcal{A}$, we call a policy i-strong if it is strong and $\mathit{Globals}(s_0^i) \subseteq \mathrm{Dom}(\pi) \cup \{s \in S^{gl} \mid s \models \gamma\}$.

When a policy is i-strong it means that the policy is strong and defined on all the global states that agent i cannot distinguish between in the initial state. It follows directly from the definition that any execution of an i-strong policy from any of those initially indistinguishable states will be successful. So if agent i comes up with an i-strong policy, then agent i knows the policy to be successful.

Example 9. The policy from above, with Bill pulling the lever to the right twice is *bill*-strong but not *anne*-strong.

Sometimes the agents cannot coordinate their plans but rather have to come up with them individually. Their policies can differ substantially, as agents often have different knowledge about the current states, applicable actions and action outcomes. To deal with agents having differing policies, we will define executions for *policy profiles*. A policy profile for a planning task Π is a family of policies $(\pi_i)_{i\in\mathcal{A}}$ where each π_i is a policy for Π. We assume actions to be instantaneous and executed asynchronously. This leads to the following definition of executions:

An execution of a policy profile $(\pi_i)_{i\in\mathcal{A}}$ is a maximal (finite or infinite) sequence of alternating global states and actions $(s_0, a_1, s_1, ...)$, such that for all $m \geq 0$,

(1) $a_{m+1} \in \pi_i(s_m)$ where $i = \omega(a_{m+1})$, and

(2) $s_{m+1} \in \mathit{Globals}(s_m \otimes a_{m+1})$.

Note that there are two different sources of nondeterminism: the nondeterminism resulting from the possibility of multiple policies prescribing actions for their respective agents (in condition 1) and the nondeterminism from nondeterministic action outcomes (in condition 2).

If all agents have one strong policy in common which all of them follow, then at execution time, the goal is guaranteed to be eventually reached. If, however, each agent acts on its individual strong policy, then the incompatibility of the individual policies may prevent the agents from reaching the goal, even though each individual policy is strong.

Bolander et al. (2018) have studied this in detail. They looked at different types of *planning agents*, which they defined as pairs (i, T), where $i \in \mathcal{A}$ is an agent name and T is a mapping from planning tasks to policies such that $T(\Pi)$ must be an i-strong policy for Π, whenever such a policy exists. The question they investigated was whether we can impose restrictions on a groups of agents $(i, T_i)_{i\in\mathcal{A}}$ so that all executions generated by this groups can be guaranteed to be successful. To simplify things, Bolander et al. (2018) have only looked at cases where all agents find *maximal strong policies* in the initial states of the planning task. These policies must be defined on all states (1) which are reachable from the initial state by arbitrary sequences of actions and (2) from which a strong policy exists. If all agents act with respect to such plans, re-planning is unnecessary and does not have to be considered.

A positive result was obtained in the general case for avoiding deadlocks (i.e., executions which end in a non-goal state where agents are waiting for each other to act). This was achieved by requiring planning agents to be *eager* and prefer own actions over other agents' actions in their plans whenever possible.

Concerning infinite executions, they have shown that there exists no type of planning agent that can prevent situations similar to the lever example. However, in cases where all agents have uniform knowledge, both deadlocks and infinite executions can be avoided if all agents are *optimally eager*, meaning if the planning agents only generate policies which are *subjectively optimal* (which means that these policies must have minimal *perspective-sensitive costs*), and that they prefer own actions over other agents' actions whenever possible without increasing the costs.

Let π be a strong policy for a planning task Π. The perspective-sensitive cost (or simply cost) of π from a state $s \in \mathrm{Dom}(\pi)$, denoted $\kappa_\pi(s)$ is defined as:

$$\kappa_\pi(s) = \begin{cases} 0 & \text{if there exists no } a \in \pi(s) \\ 1 + \max_{a\in\pi(s), s'\in \mathit{Globals}(s^{\omega(a)}\otimes a)} \kappa_\pi(s') & \text{else.} \end{cases}$$

The positive results that we get for our token-based approach will be based on the assumption that the planning agents act with respect to subjectively optimal policies.

4 Planning with Tokens

In the example from the introduction, the problem is that both agents want to pull the lever to the different goal configurations they know about, resulting in infinite executions. This problem can be eliminated by introducing a token such that only the agent that possesses the token is allowed to act. The goal of this tokenization is that only one agent gets to make a move at any time. Once the agent is done with their own actions, they can pass on the token to the next agent.

There are different ways to implement such a token. First, we have to add actions with which agents can pass the token to the next agent. Furthermore, we have to define how the first agent obtains the token. The way we decided to implement this is via an action with which any agent can take the token initially. Note that there are other possibilities. For example, we could initially assign the token randomly to one of the agents. The disadvantage with this approach is that there are planning tasks where only some but not all of the agents find a plan from their local perspectives. In these cases, assigning the token randomly would lead to an unsolvable task. We can also leave the burden of designating the first agent to act to the modeler of the planning task. But then again, to be sure to not make a solvable planning task unsolvable, the modeler would have to already know which of the agents can find a plan and which of the agents cannot.

4.1 Tokenization of Planning Tasks

We will now formally describe a function that tokenizes any given planning task. The idea is that we introduce additional predicates $\mathtt{t}_i$ for all agents $i \in \mathcal{A}$ with the intuitive meaning of "agent i possesses the token". We can then use these propositions in the preconditions of our tokenized actions. We first define the tokenization of epistemic states.

Definition 1. Let $s = (\langle W, (\sim_i)_{i\in\mathcal{A}}, V\rangle, W_d)$ be an epistemic state with proposition set P. Then the *tokenization of state* s is a new epistemic state $tok(s) = (\langle W, (\sim_i)_{i\in\mathcal{A}}, V'\rangle, W_d)$, which has the proposition set $P \cup \{\mathtt{t}_i \mid i \in \mathcal{A}\}$ and where $V' = V \cup \{\mathtt{t}_i \mapsto \emptyset \mid i \in \mathcal{A}\}$.

Furthermore, for any agent $i \in \mathcal{A}$, we define the i-tokenization $tok^i(s)$ of state s analogously, but with $V' = V \cup \{\mathtt{t}_i \mapsto W\} \cup \{\mathtt{t}_j \mapsto \emptyset \mid j \in \mathcal{A}, j \neq i\}$.

This means in $tok(s)$ no agent has the token yet and in $tok^i(s)$, the token is owned by agent i. Token ownership is always common knowledge between the agents.

We can now define the tokenization of epistemic actions.

Definition 2. Let $a = (\langle E, (\sim_i)_{i\in\mathcal{A}}, \mathrm{pre}, \mathrm{eff}\rangle, E_d)$ be an epistemic action and $i \in \mathcal{A}$ be an agent. Then

the *i-tokenization of action* a is a new epistemic action $tok^i(a) = (\langle E, (\sim_i)_{i\in\mathcal{A}}, \mathrm{pre}', \mathrm{eff}\rangle, E_d)$ with new preconditions $\mathrm{pre}'(e) = \mathsf{t}_i \wedge \mathrm{pre}(e)$ for all $e \in E$.

Since tokenized actions use the token propositions, they can only be applied to tokenized states and their successors. We can finally define the tokenization of planning tasks.

Definition 3. Let $\Pi = \langle s_0, A, \omega, \gamma\rangle$ be a planning task. We define the *tokenization of task* Π as $tok(\Pi) = \langle s_0', A', \omega', \gamma\rangle$ as follows:

- $s_0' = tok(s_0)$ where $tok(s_0)$ is the tokenization of s_0,
- $A' = \{tok^{\omega(a)}(a) \mid a \in A\} \cup \{\mathsf{takeTok}^i \mid i \in \mathcal{A}\} \cup \{\mathsf{giveTok}^{ij} \mid i, j \in \mathcal{A}, i \neq j\}$, where for all $i \neq j \in \mathcal{A}$
  - $\mathsf{takeTok}^i$ is an action consisting of a single event with precondition $\bigwedge_{k\in\mathcal{A}} \neg\mathsf{t}_k$ and effect t_i,
  - $\mathsf{giveTok}^{ij}$ is an action consisting of a single event with precondition t_i and effect $\neg\mathsf{t}_i \wedge \mathsf{t}_j$, and
- $\omega'(a') = \begin{cases} \omega(a) & \text{if } a' = tok(a) \text{ for some } a \in A \\ i & \text{if } a' = \mathsf{takeTok}^i \text{ for some } i \in \mathcal{A} \\ i & \text{if } a' = \mathsf{giveTok}^{ij} \text{ for some } i, j \in \mathcal{A}. \end{cases}$

Note that the tokenization of a planning task with proposition set P will have the proposition set $P \cup \{\mathsf{t}_i \mid i \in \mathcal{A}\}$. When talking about tokenized planning tasks, we will from now on use $S^{\mathrm{gl}}_{\mathrm{tok}}$ to denote the set of states that we get by tokenizing all the states in S^{gl}, i.e., $S^{\mathrm{gl}}_{\mathrm{tok}} = \{tok(s) \mid s \in S^{\mathrm{gl}}\} \cup \{tok^i(s) \mid s \in S^{\mathrm{gl}}, i \in \mathcal{A}\}$. Furthermore, we will use the notation $tok(A) = A'$ for the set A' of all tokenized actions from the action set, as defined above.

While we assume unit action costs in Π, it is not immediately obvious which costs we should assign to giveTok actions. Assigning them unit costs as well may be the most obvious option, but comes at the risk of introducing an unwarranted bias towards few token passings at the expense of overly costly subplans consisting of "proper" (non-token-passing) actions. On the other hand, assigning them costs of zero makes token passing free and preserves optimality. Unfortunately, with zero-cost token passings, our main theorem (Theorem 1) that states that introducing tokens prevents infinite executions, becomes invalid, since then, there can be maximal subjectively optimal i-strong policies that still lead to infinite executions, more specifically, infinite rounds of token passing. As a compromise between zero costs and unit costs, we may also assign costs of a sufficiently small $\varepsilon > 0$ to all giveTok actions. In the following, we will not study the question of those action costs further, but rather assume that all actions have unit costs.

4.2 No More Infinite Executions

Given this formalization, we can now show that we can prevent the existence of infinite executions by transforming the planning task into a tokenized planning task and requiring the agents to use subjectively optimal policies.

Theorem 1. *Let Π be a tokenized planning task and let $(\pi_i)_{i\in\mathcal{A}}$ be a profile of maximal subjectively optimal i-strong policies for Π. Then all executions of $(\pi_i)_{i\in\mathcal{A}}$ are finite.*

Proof sketch. For the case where the initial state is already a goal state and no agent takes the token, we do not have to prove anything. For the other case, let us assume that agent j has the token in some given state s with costs $\kappa_{\pi_j}(s) = c$. We can distinguish the following cases:

- If $c = 0$, then j will not perform any more action and the execution is finished.
- If $c > 0$ and $\pi_j(s) \notin \{\mathsf{giveTok}^{jk} \mid k \in \mathcal{A}\}$ then j will still have the token in each of the subjective successor states $s' \in \mathit{Globals}(s^j \otimes \pi_j(s))$ of s. By the definition of subjective costs, we have $\kappa_{\pi_j}(s') \leq c - 1$.
- If $c > 0$ and $\pi_j(s) = \mathsf{giveTok}^{jk}$ for some agent $k \in \mathcal{A}$ then agent k will have the token in each of the subjective successor states $s' \in \mathit{Globals}(s^j \otimes \pi_j(s))$ of s. By the definition of subjective costs we have $\kappa_{\pi_j}(s') \leq c - 1$. Since k plans optimally, $\kappa_{\pi_j}(s')$ must be an overestimate of $\kappa_{\pi_k}(s')$ and we thus have $\kappa_{\pi_k}(s') \leq \kappa_{\pi_j}(s') \leq c - 1$.

Since the value of c decreases by at least 1 after each action and can never fall below 0, any execution must eventually stop. □

Example 10. It is easy to see how the tokenization works with our lever example: Initially, both agents want to take the token. After one of the agents has obtained the token (which is decided nondeterministically), the agent moves the lever all the way to its goal position. The token remains with that agent and the execution is finished.

4.3 Policy Existence for Tokenized Tasks

Whereas tokenization makes sure that infinite executions disappear, unfortunately, it does not preserve all i-strong policies from the original task. An issue arises in scenarios such as the following.

Example 11. Assume that agent 1 initially holds an object and knows that either agent 2 or agent 3 desires that object, but does not know who. Without tokens, agent 1 can place the object on the table and expect the agent desiring the object to pick it up. With tokens, agent 1 can still place the object on the table, but then has to pass the token to the correct agent. Without knowing who that is, the policy cannot be tokenized in a straight forward way. Assuming that one of the agents does not even know that the other agent desires the object, giving the token to this agent would even lead to a dead-end state. In this case, while there exists a 1-strong policy for the original task, there is no 1-strong policy for the tokenized version of the task.

The underlying problem can be characterized by a property of agent 1's policy π_1: The state s in which agent 1 puts the object on the table has a global successor state s' in which one of the other agents must act, but which is indistinguishable for agent 1 to another state s'' in which the third agent must act. Formally, we have $\pi_1(s') = \{\mathsf{pickUp}\}$ and $\pi_1(s'') = \{\mathsf{pickUp}'\}$ with $\omega(\mathsf{pickUp}) \neq \omega(\mathsf{pickUp}')$.

Intuitively, to avoid this problem, we need to require that after each action of an agent i leading to some non-terminal state s', the agent i can identify a unique agent $next(i, s')$ which can act next. In the tokenized version of a policy, the

first agent would pass the token to exactly this agent or, in the case where $next(i, s') = i$, keep it.

Definition 4. We say a policy $\pi : S^{gl} \hookrightarrow \mathcal{P}(A)$ satisfies the *knows-the-next-agent* property (KNA) if there exists a function $next : \mathcal{A} \times S^{\text{gl}} \hookrightarrow \mathcal{A}$, which we call the *next agent function*, with the following property: For each state $s \in S^{gl}$, action $a \in \pi(s)$ owned by agent $i = \omega(a)$, non-terminal successor state $s' \in \mathit{Globals}(s \otimes a)$, i.e. for which $\pi(s')$ is defined and nonempty, and all states $s'' \in \mathit{Globals}(s'^i)$ which are indistinguishable to s' for i, there is an action $a' \in \pi(s'')$ such that $\omega(a') = next(i, s')$.

Note that for some policies there is more than one next agent function. Given a finite policy, it is easy to either construct a next agent function or to prove that no such function exists. This can be done by successively looking at each triple $(s, a, s') \in S^{\text{gl}} \times A \times S^{\text{gl}}$ such that $a \in \pi(s)$ and $s' \in \mathit{Globals}(s \otimes a)$. If there is an agent $i \in \mathcal{A}$ who owns for all states $s'' \in \mathit{Globals}(s'^{\omega(a)})$ an action $a' \in \pi(s'')$, we can assign it to $next(\omega(a), s')$. If there is no such agent, we know that there is no next action function and that the policy thus does not satisfy the KNA property. However, if there is always such an agent and thus the KNA property is satisfied, this means that the acting agents will be able to identify the agents to which the token can be passed next from their own perspectives and without the need of external coordination.

4.4 Tokenization of Policies

We can now tokenize *policies* which satisfy the KNA property in the obvious way: before each regular action, we add a token-passing action to the owner of the next action if necessary.

Definition 5. Let $\pi : S^{\text{gl}} \hookrightarrow \mathcal{P}(A)$ be a policy that satisfies the KNA property and let $next$ be a next agent function for π. Then we define the *tokenization of the policy* π with $next$ as $tok_{next}(\pi) : S^{\text{gl}}_{\text{tok}} \hookrightarrow \mathcal{P}(tok(A))$, where

(1) For all states $s' = tok(s) \in S^{\text{gl}}_{\text{tok}}$, i.e., in which no agent has the token, we have $tok_{next}(\pi)(s') = \{\mathsf{takeTok}^{\omega(a)} \mid a \in \pi(s)\}$.

(2) For all states $s' = tok^i(s)$, i.e., such that some agent i has the token, in the case where $next(i, s') = j \neq i$, we have $tok_{next}(\pi)(s') = \{\mathsf{giveTok}^{ij}\}$.

(3) For all states $s' = tok^i(s)$, i.e., such that some agent i has the token, in the case where $next(i, s') = i$, we have $tok_{next}(\pi)(s') = \{tok(a) \mid a \in \pi(s), \omega(a) = i\}$.

We are now ready to prove that, under the assumption of the KNA property, tokenization does indeed preserve i-strong policies.

Theorem 2. *Let π be a strong policy for a planning task Π which satisfies the KNA property and let $next$ be a next agent function for π. Then the tokenized policy $\pi' = tok_{next}(\pi)$ is also a strong policy for the tokenized planning task $\Pi' = tok(\Pi)$.*

Proof sketch. We first have to show that the policy properties are satisfied for π'. For the token taking and passing actions from the first two cases of Definition 5, the applicability condition is trivially satisfied: The token can be taken because it is still on the table, and it can be passed along because it is possessed by the correct agent. For the third case, the applicability condition is also satisfied, since all actions that are assigned to the state belong to the agent that possesses the token. Uniformity is slightly more complicated: In the first case, uniformity follows directly from the uniformity of the original policy. In the second case, uniformity follows from our definition of next agent functions which guarantees that $next(i, s') = next(i, s'')$ for all states s'' which are indistinguishable to s' for agent i. Finally, in the third case, uniformity follows for the same reason, together with the fact that all actions that were assigned to each equivalence class of indistinguishable states in the original policy are now again assigned to the corresponding states in the tokenized policy.

We can now prove that the policy is strong. We first have to show that the initial state is again either already a goal state or that it is contained in the domain of the policy. This follows directly from strength of the original policy: Either the initial state of Π is a goal state, in which case the initial state of Π' will be a goal state for Π' as well (since the tokens cannot be part of the goal formula). Or otherwise, if the initial state of Π is contained in π, its tokenized version, which is the initial state for Π' will also be contained in π'.

We then have to show that each execution of π' is successful. We first have to note that for each execution of π' in Π', there is a corresponding execution of $\pi \in \Pi$ using the untokenized actions and omitting the token taking and passing actions. Note that we cannot have any dead ends in π', because there are no dead ends in π and because our tokenization ensures that whenever there are transitions from one state to another in π, at least one of these transitions must have a corresponding transition in π' (which might include a token passing action before the actual action). Since any execution of π ends in a goal state, also any execution of π' must end in a goal state. □

5 Tokenization and Re-Planning

Note that in our analysis, so far we have always assumed that all agents act with respect to maximal strong policies which they form in the initial state of the planning task. In practice, this is a big limitation.

First of all, finding maximally strong policies can be very expensive since the state space can be huge and maximally strong policies must be defined for all reachable states from which a strong policy exists. Even in decidable fragments of DEL planning it stands to reason that finding maximally strong plans can be much more difficult than finding optimal plans. Also, we cannot deal with situations yet where only one of the agents has an i-strong policy from the start, and where the other agents will find their plans and join taking part in the execution only after a few actions of agent i, which is one of the main advantages of i-strong policies.

An alternative to using maximal strong policies is to employ a *re-planning* regime. This approach has been taken by Nebel et al. (2019) in the specialized setting of *multi-agent path finding with destination uncertainty*, which has been

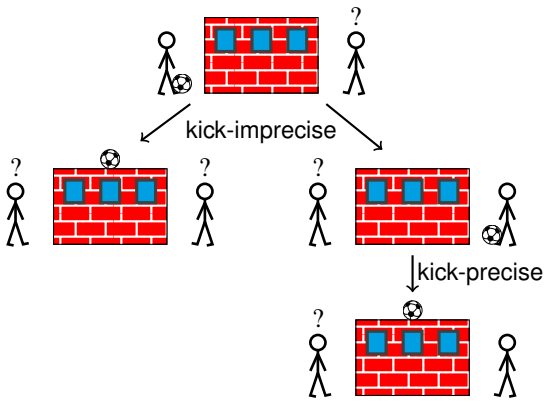


Figure 2: Two agents, a ball and a roof.

originally formalized as DEL planning task (Bolander et al. 2018). Nebel et al. showed that even in this restricted class of implicit-coordination problems, and using a re-planning regime, optimally eager agents can still produce infinite executions. In the following, we formalize re-planning executions for the general case and analyze whether tokenization helps to prevent infinite executions.

Definition 6. A *re-planning execution* of a group of planning agents $(i, T_i)_{i\in\mathcal{A}}$ is a maximal (finite or infinite) sequence of alternating global states and actions $(s_0, a_1, s_1, \ldots)$, such that for all $m \geq 0$,

(1) $a_{m+1} \in T_i(\Pi|_{s_m})(s_m)$ where $i = \omega(a_{m+1})$, and
(2) $s_{m+1} \in \mathit{Globals}(s_m \otimes a_{m+1})$,

where $\Pi|_{s_m}$ is the same planning task as Π, but where the initial state has been replaced by s_m. We call such an execution successful if it is a finite execution $(s_0, a_1, s_1, \ldots, s_n)$ such that $s_n \models \gamma$.

Unfortunately, Theorem 1 can not be transferred directly to re-planning agents. Consider the following counter-example, as depicted in Figure 2.

Example 12. There are two agents standing at opposing sides of a building. Since the building is between them, they cannot see each other. The goal of the planning task is that a football, which is initially on the side of the left agent, ends up on top of the building. The agents can observe the ball only if it is on their respective side of the building. In our depiction, an agent that has a question mark on the top of its head is unsure whether the ball is already on the roof or with the other agent. Both agents have an action to kick the ball. However, the left agent cannot aim as accurately as the right agent. If he kicks the ball, it might either land on the roof or on the side of the other agent. The right agent has sufficient aim, so after she kicks the ball, she will be certain that the ball is on the rooftop. The left agent will not know when this happens, since he cannot see the shot.

Figure 2 shows a policy that is i-strong for both agents. Interestingly, it contains a goal state in which both agents do not know that the goal is satisfied. Since the figure is also a depiction of the entire state space of the planning task (and since in the goal states, no agent can act), it is clear that re-planning agents who act with respect to i-strong policies will always generate successful executions.

However, in the tokenization of the planning task, the fact that the agents now do re-planning leads to an infinite execution in the case where the imprecise kicking action of the left agent already succeeds landing the ball on the roof. This is because the agent must pass the token to the agent on the right to cover for the contingency where the ball went over the roof. The agent on the right must then re-plan and cover for the contingency where the ball is still with the left agent and pass the token directly back, which leads to the same state as before. Thus the agents will effectively pass the token back and forth.

The reason for the infinite execution is that in the end, none of the agents knows that the goal has been reached, and both of the agents consider it possible that the other agent still has some work to do. If we allowed the agents to do some kind of forward induction, i.e., reasoning about the motive behind the other agent's action, they could maybe infer from the fact that the token has been passed to them that the ball must already be on the roof. However, it is unclear how a reasonable solution concept for implicit coordination with forward induction would look like and this is a major topic for future research.

A more direct way to prevent such situations and which works with our existing notion of strong policies is to ensure that each policy is *stable*, in the sense that in terminal states, all agents who have applicable actions know that the goal has already been reached.

Definition 7. We say that a policy $\pi : S \hookrightarrow \mathcal{P}(A)$ is *stable* for a goal γ, if for all states $s \in S$, actions $a \in \pi(s)$ and successor states $s' \in \mathit{Globals}(s \otimes a)$ for which there is no action $a' \in \pi(s')$ (i.e., terminal states), we have $s' \models K_i\gamma$ for all agents i who have actions that are applicable in s'^i.

Although this approach seems to be very restrictive in general, it makes a lot of sense for tokenized planning tasks because in any given state of an execution, with the exception of the initial state, there is only one agent that has applicable actions anyway. We can now verify that there are no infinite executions for tokenized planning tasks, given all agents are optimally eager and produce only stable plans.

Theorem 3. *Given a tokenized planning task $tok(\Pi)$ with goal γ and a group $(i, T_i)_{i\in\mathcal{A}}$ of optimally eager planning agents, such that for arbitrary states $s \in S^{gl}_{tok}$, the agents produce only policies $T_i(tok(\Pi)|_s)$ which are stable for γ. Then all re-planning executions of $(i, T_i)_{i\in\mathcal{A}}$ are finite.*

Proof sketch. The proof works analogously to the proof of Theorem 1, with the difference that agents can have a different policy for each state. The first two cases are identical: If the agent who has the token finds a policy with cost 0, the execution is finished. And if the agent who has the token has an action that is not a token passing action, then in the successor state he will have the token again and be guaranteed to find a policy where the cost decreases by at least 1.

The difference is in the third case where the token is passed from an agent j to another agent k. Here, the proof of Theorem 1 relies on the assumption that if agent k wants

to apply an action after obtaining the token, then this action must also be part of a policy that is subjectively optimal from the initial state. For example, in the rooftop example with tokens, there is no policy starting from the initial state such that if the ball lands on the roof immediately and the token is passed to the right agent, this agent will try to give the token back (because this would create a cycle in the policy). Instead, the execution would stop with the agent not knowing that the goal has been reached. With re-planning and without the stability property, the right agent would find a new plan in which the token is given back and then, if the goal has already been reached, the left agent does nothing, or otherwise the policy proceeds as in Figure 2. However, in the execution, after the token has been given back, the left agent would also re-plan and pass the token straight back.

We will now look at what happens with policies satisfying our stability property. As before, we assume that the token is passed from agent j to k, leading from a state s to a state s' in which only agent k has applicable actions. We distinguish between the following cases:

- If $\pi_j = T_j(tok(\Pi)|_s)$ assigns some action to state s', this action must be owned by agent k and it is thus clear that π_j must also be a k-strong policy for s'. Since agent k plans optimally, $\pi'_k = T_k(tok(\Pi)|_{s'})$ will be a policy with $\kappa_{\pi'_k}(s') \leq \kappa_{\pi_j}(s') \leq \kappa_{\pi_j}(s) - 1$.
- If π_j assigns no action $a' \in \pi_j(s')$ to s', then because of the stability condition (since at least the token passing actions are applicable), agent k must know that the goal is satisfied in s' and thus $\pi'_k = T_k(tok(\Pi)|_{s'})$ will be a policy with $\kappa_{\pi'_k}(s') = 0$.

Since, as before, the cost of the policy belonging to the agent who acts in each state of the execution decreases in each step by at least 1 and can never fall below 0, any execution must eventually stop. □

Instead of defining a new agent type that also requires generated policies to be stable, we can modify the planning tasks to enforce that each strong policy for the planning task must be stable. This is especially easy if the planning task is already tokenized: If the propositions $\mathtt{t}_i$ denote that an agent i has the token, we can simply change the goal formula from γ to $\gamma' = \gamma \wedge \bigwedge_{i \in \mathcal{A}}(\mathtt{t}_i \rightarrow K_i\gamma)$. Thus, in each new goal state, an agent who has the token must know that γ is satisfied and, by introspection, also that γ' itself is satisfied. Therefore, any strong policy trivially satisfies the stability condition, and thus agents never have reason to pass on the token to another agent in goal states.

Example 13. Imagine that in our football example, the left agent has an additional action using which also he can pass the ball to the right agent with certainty, e.g., by throwing the ball instead of kicking it. Then, if we encode our stability condition into the goal formula, and given both agents are optimally eager, a successful execution is guaranteed. The left agent will throw the ball to the right agent who will then kick it onto the rooftop. Without the stability condition, the left agent might just as well decide to kick the ball, which might result in both agents passing the tokens back and forth, as seen previously.

6 Conclusion

We have looked at how the tokenization of epistemic planning tasks can be used to mitigate the problem of infinite executions. We have shown that in the case of agents acting with respect to maximally strong policies, tokenization successfully prevents infinite executions. However, there are planning tasks for which, despite the existence of a strong policy, there is no strong policy for the tokenized task. The strong policies of the original task which can be tokenized are characterized by the knows-the-next-agent property.

So far, we have considered only a tokenization in which the token is passed directly from one agent to another agent. This is a big restriction since we can easily construct planning tasks where the first agent to act knows that one of the other agents can finish the plan, but not which of the agents. Even if for this task infinite executions are not possible, the fact that no policy satisfies the KNA property implies that there are no strong policies for the tokenization. A possible remedy could be to allow the agents to pass the token to multiple agents at once, one of which then has to decide to take the next action and continue passing the token. However, without additional constraints (e.g., on the subsets of agents that the token can be passed to), using such a tokenization we can easily end up with infinite executions again. This is because after each action the token can be passed back to all of the agents, and thus each strong policy of the original task also corresponds to a strong policy for the tokenization. Thus, if we can get infinite executions for the original task, we can also get infinite executions for this kind of tokenization. Note that the solution policies for most of the tasks considered in literature, including multi-agent path finding with destination uncertainty, satisfy the KNA property meaning that the simple tokenization from this paper is sufficient to solve these tasks while avoiding infinite executions.

Importantly, if there is a policy for the original planning task that can be tokenized, it can be found by directly by planning for the tokenized task, which is not a more complex problem than planning for the original task. This is because the tokenization increases the size of the input task only linearly. Note that while in the general case the strong policy existence problem is undecidable, there exist tractable fragments (Engesser and Miller 2020).

We have furthermore shown that for re-planning agents, tokenization alone is not sufficient to prevent infinite executions. However, we can employ an additional stability condition that can be encoded into the goal formula.

For future work, we plan to investigate tokenized planning tasks in the context of forward induction. This would mean allowing the agents to infer knowledge by reasoning about the motive of other agents' actions. For example, an agent could pass the token to another agent to signal that that agent has an action available which progresses towards the goal, even if the agent does not know that yet. However, forward induction has not been formalized in the context of epistemic planning for implicit coordination so far. Supporting such reasoning would arguably require a solution concept that goes beyond what is possible with strong policies.

References

Blackburn, P.; de Rijke, M.; and Venema, Y. 2001. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Bolander, T.; Engesser, T.; Mattmüller, R.; and Nebel, B. 2018. Better eager than lazy? How agent types impact the successfulness of implicit coordination. In *Proceedings of KR 2018*, 445–453.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1–2):35–84.

Engesser, T., and Miller, T. 2020. Implicit coordination using FOND planning. In *Proceedings of AAAI 2020*, 7151–7159.

Engesser, T.; Bolander, T.; Mattmüller, R.; and Nebel, B. 2017. Cooperative epistemic multi-agent planning for implicit coordination. In *Proceedings of M4M 2017*, 75–90.

Huang, X.; Fang, B.; Wan, H.; and Liu, Y. 2017. A general multi-agent epistemic planner based on higher-order belief change. In *Proceedings of IJCAI 2017*, 1093–1101.

Karpas, E.; Shleyfman, A.; and Tennenholtz, M. 2017. Automated verification of social law robustness in STRIPS. In *Proceedings of ICAPS 2017*, 163–171.

Kominis, F., and Geffner, H. 2015. Beliefs in multiagent planning: From one agent to many. In *Proceedings of ICAPS 2015*, 147–155.

Le, T.; Fabiano, F.; Son, T. C.; and Pontelli, E. 2018. EFP and PG-EFP: epistemic forward search planners in multi-agent domains. In *Proceedings of ICAPS 2018*, 161–170.

Loucks, L. K., and Shaheen, A. A. 1997. System and method for multi-level token management for distributed file systems. US Patent 5,634,122.

Makki, K.; Banta, P.; Been, K.; Pissinou, N.; and Park, E. K. 1992. A token based distributed K mutual exclusion algorithm. In *Proceedings of SPDP 1992*, 408–411.

Maubert, B.; Pinchinat, S.; and Schwarzentruber, F. 2019. Reachability games in dynamic epistemic logic. In *Proceedings of IJCAI 2019*, 499–505.

Muise, C.; Belle, V.; Felli, P.; McIlraith, S.; Miller, T.; Pearce, A. R.; and Sonenberg, L. 2015. Planning over multi-agent epistemic states: A classical planning approach. In *Proceedings of AAAI 2015*, 3327–3334.

Nebel, B.; Bolander, T.; Engesser, T.; and Mattmüller, R. 2019. Implicitly coordinated multi-agent path finding under destination uncertainty: Success guarantees and computational complexity. *Journal of Artificial Intelligence Research* 64:497–527.

Nir, R., and Karpas, E. 2019. Automated verification of social laws for continuous time multi-robot systems. In *Proceedings of AAAI 2019*, 7683–7690.

van Ditmarsch, H.; van der Hoek, W.; and Kooi, B. 2007. *Dynamic Epistemic Logic*, volume 337 of *Synthese Library*. Springer.

Query Content in Sequential One-shot Multi-Agent Limited Inquiries when Communicating in Ad Hoc Teamwork

William Macke[1], **Reuth Mirsky**[1], **Peter Stone**[1, 2]
[1] The University of Texas at Austin, [2] Sony AI
{wmacke,reuth,pstone}@cs.utexas.edu

Abstract

Communication in Ad Hoc Teamwork (CAT) is a research area that investigates how communication can be leveraged by an agent that plans in a distributed, multi-agent collaborative environment, even if that agent does not have knowledge about its teammates or their plans a priori. This paper reports our progress in identifying three factors that can impact the complexity of CAT – environment, teammates, and communication protocol. Following the identification of these components, this paper investigates three extensions from existing work that affect each of these factors respectively – richer environments, complex teammate representations, and complex communication protocols. We present new algorithms to compute when to query under these new configurations, as well as preliminary results of their performance.

Introduction

Autonomous agents are becoming increasingly capable of solving complex tasks, but encounter many challenges when required to solve such tasks as a team. For example, service robots have been deployed to assist medical teams in the recent pandemic outbreak. Such robots' coordination strategy cannot be learned or decided a priori, as it interacts with previously unmet teammates (Cakmak and Thomaz 2012). This motivation is the basis for *ad hoc teamwork*, which is defined as collaborating with teammates without pre-coordination (Stone et al. 2010; Albrecht and Stone 2018). This terminology reflects that the *collaboration* is ad hoc – the ways in which the agents learn, act, and interact may be quite principled. Our previous work on CAT identified a specific variant of CAT, namely the Sequential One-shot MultiAgent Limited Inquiry CAT scenario, or SOMALI CAT (Mirsky et al. 2020). In SOMALI CAT, the agents execute *sequential plans* and only the ad hoc agent can inquire about a teammate's goal. SOMALI CAT was defined to be a broadly representative class of CAT problems. In such a SOMALI scenario, the robot can fetch different tools for a physician in a hospital. The physician would normally prefer to avoid the additional cognitive load of communicating with the robot, but will answer an occasional question from it so that the robot can be a better collaborator. The results from this work were evaluated on a simulated test-bed, namely *the tool fetching domain*. The algorithm presented in that work was a means to decide when to query in such a SOMALI scenario. This paper reports our progress investigating the different factors that can affect the complexity of a SOMALI scenario: environment, teammates, and communication protocol. An additional contribution is a set of heuristic algorithms for choosing when to query, that are shown to outperform previous work in these complex configurations.

Background

Communicating agents has been a fertile research area in the context of distributed multiagent systems (Singh 1998; Cohen, Levesque, and Smith 1997; Decker 1987). Goldman and Zilberstein (2004) formalized the problem of a decentralized POMDP with communication (DEC-POMDP-com). Communication in Ad-Hoc Teamwork (CAT) is a close problem that shares some similar assumptions: all teammates strive to be collaborative and the agents have a predefined communication protocol available. However, DEC-POMDP-com uses a single model that is collaboratively controlled by multiple agents, whereas CAT is set from the perspective of one agent that has no additional knowledge about its teammates' policies and that it cannot change the properties of these teammates (Stone et al. 2010).

Barrett et al. (2014) considered a scenario in which either teammates are assumed to share a common communication protocol, or else this assumption can be quickly tested on the fly (e.g. by probing). Their work was situated in a very restrictive multi-agent setting, namely a multi-arm bandit, where each task was a single choice of which arm to pull. A different type of CAT scenarios refers to tasks where a single agent reasons about the sequential plans of other agents, and can gain information by querying its teammates or by observing their actions (Mirsky et al. 2020). This Sequential One-shot Multi-Agent Limited Inquiry CAT scenario, or SOMALI CAT, was inspired by the use case of a service robot that is stationed in a hospital, who mainly have to retrieve supplies for physicians or nurses, and has two main goals to balance: understanding the task-specific goals of its human teammates, and understanding when it is appropri-

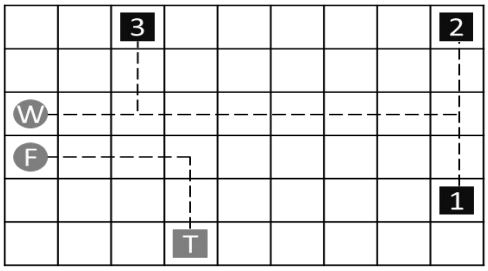


Figure 1: Example of the tool fetching domain. W, F, and T are the locations of the worker, fetcher, and toolbox respectively. The locations of the workstations are represented by the numbered squares.

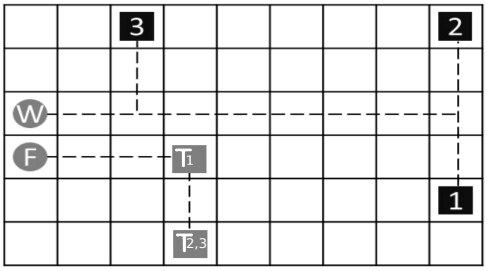


Figure 2: Example of a domain with multiple toolboxes. T_1 houses tool 1 while $T_{2,3}$ houses tools 2 and 3.

ate to ask questions over acting autonomously. In such SOMALI CAT scenarios, we have additional assumptions: the task performed requires a sequence of actions; it is also an episodic, one-shot task; the environment is static, deterministic, and fully observable; the teammate is assumed to have perfect knowledge about the environment; the teammate is assumed to plan optimally, given that it is unaware of the ad hoc agent's plans or costs; and there is one communication channel, where the ad hoc agent can query as an action, and if it does, the teammate will forgo its action to reply truthfully (the communication channel is noiseless).

The Tool Fetching Domain Our previous work in SOMALI CAT introduced an experimental domain known as the tool fetching domain. This domain consists of an ad hoc agent, the *fetcher*, attempting to meet a teammate, the *worker*, at some workstation with a tool. The worker needs a specific tool depending on which station is its goal, and the worker's goal is unknown before hand to the fetcher. It is the job of the fetcher to deduce the goal of the worker based on its actions, and to bring the correct tool to the correct workstation. At each timestep, the agents can execute one action each. Additionally, the fetcher can query the worker with questions of the form "Is your goal one of the stations $g_1, g_2 ... g_N$?", where $g_1, \ldots, g_N \subseteq G$ is a subset of all workstations. All queries in the original setup are assumed to have a cost identical to moving one step, regardless of the content of the query. Figure 1 shows an example of this domain where the fetcher is following the path to the toolbox while the worker is following one of the paths to an unknown workstation. In this paper, the domain was implemented as a custom OpenAI Gym environment (Brockman et al. 2016).

When to Query To reason about when to act in the environment and when to query, three different reasoning zones were defined for each query that the ad-hoc agent can ask to disambiguate a subset of goals ($G' \subset G$) from ($G \setminus G'$):

Zone of Branching (Z_B) for a set of goals $G' \subseteq G$ is the set of timesteps from when the ad hoc agent (the fetcher) is required to commit to a specific goal and until the end of the episode, which means the timesteps in which it might take a different action from the one it would have taken if it had perfect knowledge about the teammate's true goal.

Zone of Information (Z_I) for a set of goals $G' \subseteq G$ is the set of timesteps from the beginning of the plan and until there is no longer any ambiguity in the domain between goals in G' and $G \setminus G'$.

Zone of Querying (Z_Q) for a subset of goals $G' \subseteq G$ is the intersection of these two sets of timesteps, where there may be a positive value in querying instead of acting.

Given these zones for each subset of goals G', we can identify the **Critical Querying Point (CQP)** as the first timestep inside $Z_Q(G')$, and is the first timestep in which the ad hoc agent should consider whether to query "Is your goal one of the stations in G'?". If $Z_Q(G')$ is empty, then there is no time in which this query can be useful and $CQP(G') = -1$. In Figure 1, some of the critical querying points are $CQP(\{1\}) = CQP(\{2\}) = 6$, as it takes the fetcher 5 timesteps to reach the toolbox and only then it enters Z_B for goals 1 and 2.Notice that $CQP(\{1,3\}) = 6$ as well, as in this case $G' = \{1,3\}$ and $G \setminus G' = \{2\}$, which means that there is still a benefit from disambiguating a group of goals that contains goal 1 and a group of goals that contains goal 2. $CQP(\{3\}) = -1$, as by the time the fetcher reaches the toolbox, the worker has already reached or passed station 3.

Generalizing SOMALI CAT

While previous work successfully demonstrated that each set of possible queries had a unique optimal time to ask, namely the CQP, it used a naive approach of choosing half the relevant goals at random for deciding the *content* of a query when multiple queries share the same CQP. Such an approach also misses the potential in more complicated scenarios, such as when the worker chooses its goal with a non-uniform probability. or when different queries cost different amounts based on their content. In this section we modify some of the assumptions from previous work: having multiple tool stations instead of just one (hence having multiple zones of branching); having a non-uniform distribution over

the possible goals the worker might reach; and having different query cost models rather than a unit cost per query.

Multiple Zones of Branching In previous work, domain setups only contained one toolbox that housed all the tools. This meant that effectively all queries had the same CQP regardless of their content, so an efficient strategy was to query about half of the goals randomly at the CQP. However, when there is more than one tool location present, this strategy no longer holds. Figure 2 shows an example when this strategy falls short. In this example, the tool for station 1 is in the top toolbox (T_1) and the tools for goals 2 and 3 are in the bottom toolbox (T_2, T_3). The first point in time when the fetcher might want to query is after it arrives at the toolbox T_1. If it were to query about half the goals randomly, it may ask "Is your goal station 1?" This action is effectively a wasted query, since it does not add new information, which means that the fetcher cannot act upon its current knowledge.

Non-Uniform Goal Distribution Another assumption made in previous work was that the worker is always assigned a goal according to a uniform probability distribution. This assumption may not hold in practice. For instance, if a human were to take the part of the worker, they may be more likely to choose goals that are closer to them. Knowledge of this distribution may allow the fetcher to construct more informative queries. For instance, if there are three goals in a domain, and the worker goal distribution is $g_1 = 0.98, g_2 = 0.01, g_3 = 0.01$ then it is likely better to query about $\{g_1\}$ or $\{g_2, g_3\}$ than about $\{g_2\}$ or $\{g_3\}$ (the fetcher is more likely to learn the worker's true goal with the former queries than with the latter ones).

Query Cost An assumption that was used in previous work is that the cost of querying is uniform, relatively small, and is not affected by the content of the query or its timing. However, it is very likely that a different cost model would result in different performance. If the cost of a query is larger than the value of the information gained, then there will be no benefit from asking such a query. We investigate how effective various query strategies are with different cost models. We consider two variables for a cost model in particular: *base cost (bc)*, or the initial cost of asking any query, and *station cost (sc)*, or the additional cost of including another station in a query. The total cost of asking a query q that asks "Is your goal one of the stations $g_1, g_2, ...g_N$?" is $bc{+}sc{*}N$. Importantly, in such a cost model, querying about many goals is more costly than querying about just one goal. This means that in the first running example, querying about $\{g_2, g_3\}$ or $\{g_1\}$ are no longer equal in their potential benefit, and the latter, smaller query becomes preferable.

Query Algorithms

We present a basic objective for determining what to query without any of the new assumptions presented earlier in this section. Previous work would query about half the relevant goals to try and maximize information gain. However reasoning more thoroughly regarding which goals to ask about can give even better performance. Consider the pairs of goals $G_B = \{(g_i, g_j) | t \in Z_B(g_i, g_j)\}$ where t is the current time and $g_i, g_j \in G_B$, a set of all N possible goals that might still be the true goal of the worker. We can construct a binary vector $\vec{x}$ of length N such that if x_i is the i-th value in that vector, then goal g_i is included in the query if and only if $x_i = 1$. A query that asks about a subset of goals $G' \subseteq G$ will disambiguate between the sets G' and $G \backslash G'$. Therefore, to increase the information gained from the query, we want to split the pairs of goals (g_i, g_j) as evenly as possible between G' and $G \setminus G'$. We write the following maximization goal

$$\max \sum_{(g_i, g_j) \in G_B} (x_i \oplus x_j) \tag{1}$$

The term in the objective is 1 only when one x is 0 and the other is 1. This objective ensures that the query disambiguates between as many of the pairs of goals as possible. Consider the example above when the fetcher arrives on toolbox T_1. This approach is guaranteed to now ask either "Is your goal in $\{g_1\}$?" or "Is your goal in $\{g_2, g_3\}$?", both of which are guaranteed to allow the fetcher to act in the next timestep regardless of the worker's answer.

While the above can easily handle multiple tool locations, it can fail under circumstances where there's a non-uniform probability distribution over the worker's possible goals. To reason about such circumstances, we modify the integer program's objective from the previous section to weigh the goals by the ad-hoc agent's current belief state:

$$\max \sum_{(g_i, g_j) \in G_B} (P(g_i) + P(g_j)) * (x_i \oplus x_j) \tag{2}$$

where $P(g_i)$ refers to the probability that the worker's goal is g_i. Intuitively, this new equation will prioritize goals that are more likely. If the worker's goal has the same probability to be either g_i or g_j, than it is most informative to disambiguate between the two. On the other hand, if one goal has a probability of 0.99, as in the example in section , it would be advantageous just to query about that one goal. Incorporating the probabilities of goals in the objective as shown above results in the method prioritizing disambiguating this higher probability goal from others.

Finally, a complete model that is able to reason about all of the extensions presented in the previous section is still required to incorporate different query cost models. Since we want to minimize the needed query cost as part of our objective within the integer program, we add the negative cost of the query to our objective. Consequently, the final integer program objective becomes

$$\max \sum_{(g_i, g_j) \in G_B} (x_i \oplus x_j) * (P(g_i) + P(g_j)) - \sum_i (x_i * sc) \tag{3}$$

where *sc* is the cost of including a goal in a query. This objective now simultaneously attempts to maximize the probability that the ad hoc agent will be able to act in the next timestep and minimize the cost of the query. All objectives shown above were solved with the Coin-Or Integer Program Solver using PuLP as the front end (Forrest et al. 2018; Mitchell, Consulting, and Dunning 2011).

Table 1: The different algorithms used in the experiments.

Never Query	Never Queries but waits until it knows an action is optimal
Random Query	Randomly asks about half the remaining potential goals when in a Z_Q
Max Binary Policy	Optimizes the query according to Equation 1 when in a Z_Q
Goal Prob Policy	Optimizes the query according to Equation 2 when in a Z_Q
Weighted Cost Policy	Optimizes the query according to Equation 3 when in a Z_Q

Results

We hypothesized that our new methods should be able to significantly outperform the previous approach regardless of the cost model used or the worker's probability of choosing goals. The experiments compare 5 different query algorithms, as presented in Table 1. Additionally, if the fetcher is going to query in a given timestep, it may be advantageous to query about goals that are not critical for acting. That is, goals g such that $(g, g') \notin G_B \forall g' \in G$. While it is possible to modify the objective to consider these goals, the size of the integer program quickly increases beyond what is reasonably tractable. Therefore, as a heuristic, we include half of these stations in the query in order to increase information gain. Each of the following experiments has a grid size of 50×50 with 100 stations located randomly. Each station has a required tool that is in one of five random toolbox locations. We assume the cost of all non-querying actions is 1. All results are averaged over 100 random instances where each instance consists of a station and tool locations, the initial fetcher and worker positions and a specific workstation assigned as the worker's goal.

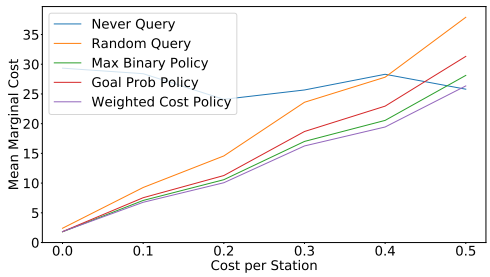


Figure 3: Marginal cost of the fetcher's plan execution given a varying cost per station in a query, under *uniform* distribution of goals for the worker.

Figure 3 shows the marginal plan execution costs over the optimal plan, assuming an oracle that lets the fetcher know what's the worker's true goal. The x-axis shows different additional cost per workstation (sc) in a query, given an initial query cost (bc) of 0.5. The probability of a worker being assigned a goal is *uniform across all 100 goals*. As shown, all query methods decrease in performance as the cost per station increases, however Weighted Cost Policy decreases at a much lower rate compared to other methods and never performs significantly worse than the Never Query method.

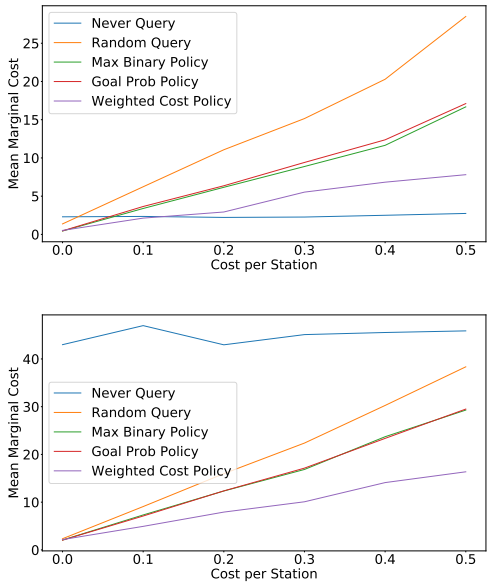


Figure 4: Marginal cost of the fetcher's plan execution when the probability of a worker being assigned to a goal is a softmax of the worker's negative (top) and positive (bottom) distances to a goal.

Figure 4 show marginal costs with various additional cost per workstation in a query, but with a *non-uniform worker goal distribution*. The probability of a worker being assigned a goal as the softmax of the negative and of the positive worker's distances to goals respectively. Intuitively, it respectively defines workers that are most likely to prefer workstations that are closer or farther from their initial location. These graphs present similar results, with the primary difference being the relative performance of the Never Query Strategy. In Figure 4 (top), the worker is much more likely to move to a close workstation, which reduces the maximum time before the fetcher knows the worker's goal. Similarly in Figure 4 (bottom), the worker is more likely to move to a distant station, increasing this maximum time for the fetcher to know its goal, which causes the Never Query strategy to perform better or worse respectively.

Conclusion

We presented several extensions to SOMALI CAT and several novel algorithms for determining what and when to query, and demonstrated their performance in the Tool Fetching Domain. Our new algorithms were able to outperform previous techniques in multiple scenarios. For future work we plan to further generalize our query algorithms to perform well in any SOMALI CAT domain, regardless of the query cost model, probability over goals, or other domain-specific details.

ACKNOWLEDGMENTS

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (CPS-1739964, IIS-1724157, NRI-1925082), ONR (N00014-18-2243), FLI (RFP2-000), ARO (W911NF-19-2-0333), DARPA, Lockheed Martin, GM, and Bosch. Peter Stone serves as the Executive Director of Sony AI America and receives financial compensation for this work. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research. Studies in this work were approved under University of Texas at Austin IRB study numbers 2015-06-0058 and 2019-03-0139.

References

Albrecht, S. V., and Stone, P. 2018. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence* 258:66–95.

Barrett, S.; Agmon, N.; Hazon, N.; Kraus, S.; and Stone, P. 2014. Communicating with unknown teammates. In *AAMAS*, 1433–1434.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym.

Cakmak, M., and Thomaz, A. L. 2012. Designing robot learners that ask good questions. In *ACM/IEEE international conference on Human-Robot Interaction*.

Cohen, P. R.; Levesque, H. J.; and Smith, I. A. 1997. On team formation. *Synthese Library* 87–114.

Decker, K. S. 1987. Distributed problem-solving techniques: A survey. *IEEE transactions on systems, man, and cybernetics* 17(5):729–740.

Forrest, J.; Ralphs, T.; Vigerske, S.; LouHafer; Kristjansson, B.; jpfasano; EdwinStraver; Lubin, M.; Santos, H. G.; rlougee; and Saltzman, M. 2018. coin-or/cbc: Version 2.9.9.

Goldman, C. V., and Zilberstein, S. 2004. Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of artificial intelligence research* 22:143–174.

Mirsky, R.; Macke, W.; Wang, A.; Yedidsion, H.; and Stone, P. 2020. A penny for your thoughts: The value of communication in ad hoc teamwork. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Mitchell, S.; Consulting, S. M.; and Dunning, I. 2011. Pulp: A linear programming toolkit for python.

Singh, M. P. 1998. Agent communication languages: Rethinking the principles. *Computer* 31(12):40–47.

Stone, P.; Kaminka, G. A.; Kraus, S.; and Rosenschein, J. S. 2010. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *AAAI*.

Partial Disclosure of Private Dependencies in Privacy Preserving Planning

Rotem Lev Lehman[1] and **Guy Shani**[1] and **Roni Stern**[1,2]
[1]Software and Information Systems Engineering, Ben Gurion University of the Negev, Be'er Sheva, Israel
[2]Palo Alto Research Center, Palo Alto, CA, USA
levlerot@post.bgu.ac.il, shanigu@bgu.ac.il, sternron@post.bgu.ac.il

Abstract

In collaborative privacy preserving planning (CPPP), agents can plan together by revealing private dependencies between their public actions to other agents. Perhaps one of the best methods for computing plans under privacy constraints uses a projection of the complete problem that captures these private dependencies. In this paper we investigate the partial disclosure of such private dependencies. We create a projection where agents publish only a part of their dependencies, and attempt to create a complete plan using these dependencies only. We investigate different strategies for deciding which dependencies to publish, and how they affect both the coverage and the privacy leakage of the solutions. Experiments over standard CPPP domains show that the proposed dependency-sharing strategies enable creating an effective projection without sharing all private dependencies.

1 Introduction

Designing autonomous agents that act collaboratively is an important goal. A fundamental requirement of such collaboration is to plan for multiple agents acting to achieve a common set of goals. *Collaborative Privacy-Preserving Planning (*CPPP*)* is a multi-agent planning task in which agents need to achieve a common set of goals without revealing certain private information (Brafman and Domshlak 2008). In particular, in CPPP an individual agent may have a set of private facts and actions that it does not share with the other agents. CPPP has important motivating examples, such as planning for organizations that outsource some of their tasks.

There are two common approaches to CPPP: *single search* and *two-level search*. Single search solvers operate by running a joint forward search (Nissim and Brafman 2014; Štolba and Komenda 2017) where agents that apply public actions send the resulting state to other agents that continue the forward search. Two-level search solvers operate by creating a public plan that is shared by all agents, and then have each agent extend it locally with private actions. In either case, the agent publish *dependencies* between the public actions. For example, in a solver from the first approach, an agent i that receives a state s from another agent j that applied a public action a_1, also receives from j an index for its own private facts. Later, when i continues the search from s and executes another public action a_2, it returns the state to j with the same index. Thus, j learns that applying a_1 may have helped i in making a_2 possible. We call this a *private dependency* between the actions.

Maliah, Shani, and Stern (2016a) take this idea further, and compute and publish a set of such private dependencies in the form of artificial facts. This allows the agents to jointly create and publish a *projection* of the problem that contains all the public facts, public actions, and published artificial facts that capture private dependencies. Such a projection can be used to construct heuristics for single search CPPP algorithms such as MAFS. In two-level search solvers, this projection can be used to generate the public plan, defining the public plan to be a solution to the problem defined by the projection.

In many cases, however, the agents can construct a plan requiring only a small portion of the private dependencies. In such cases, it may be preferable to reveal only a part of the dependencies, intuitively reducing the amount of disclosed private information. In this paper we focus on the settings where agents publish only a limited number of such dependencies. We suggest 4 different methods for deciding which dependencies should be published first, in order to construct a plan with as little disclosed dependencies as possible.

We provide experiments on standard benchmarks from the CPPP literature, showing that our methods, in many domains, publish as little private dependencies as possible in order to reach a plan. We also analyze the privacy leakage of our methods (Štolba, Tožička, and Komenda 2018), showing that, as one would intuitively expect, publishing more dependencies leaks more private information.

2 Background

A MA-STRIPS problem (Brafman and Domshlak 2013) is represented by a tuple $\langle P, \{A_i\}_{i=1}^{k}, I, G\rangle$ where: k is the number of agents, P is a finite set of primitive propositions (facts), A_i is the set of actions agent i can perform, I is the start state, and G is the goal condition.

Each action $a = \langle pre(a), \mathit{eff}(a)\rangle$ is defined by its preconditions ($pre(a)$), and effects ($\mathit{eff}(a)$). Preconditions and effects are conjunctions of primitive propositions and literals, respectively. A state is a truth assignment over P. G is a conjunction of facts. $a(s)$ denotes the result of applying

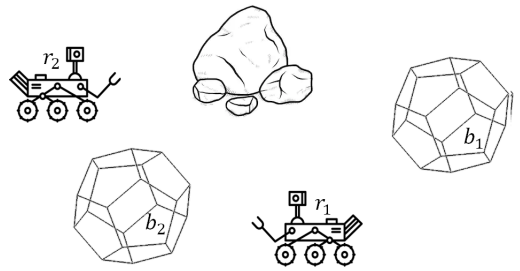


Figure 1: The rovers domain, where two rovers, r_1 and r_2 can access two base stations b_1 and b_2, collaborating to take measurements of a rock.

action a to state s. A *plan* $\pi = (a_1, \ldots, a_k)$ is a solution to a planning task iff $G \subseteq a_k(\ldots(a_1(I)\ldots)$.

Privacy-preserving MA-STRIPS extends MA-STRIPS by defining sets of variables and actions as private, known only to a single agent. $private_i(P)$ and $private_i(A_i)$ denote the variables and actions, respectively, that are private to agent i. $public(P)$ is the set of public facts in P. $public_i(A_i)$, the complement of $private_i(A_i)$ w.r.t. A_i, is the set of public actions of agent i. Some preconditions and effects of public actions may be private, and the action obtained by removing these private elements is called its *public projection*, and it is known to other agents. When a public action is executed, all agents are aware of the execution, and view the public effects of the action. The goals can be public, but can also be private to a single agent. An agent is aware only of its *local view* of the problem, that is, its private actions and facts, its public actions, the public facts, and the public projection of the actions of all other agents. That is, for public actions of other agents, the agent's local view contains only the public preconditions and effects of these actions.

In the Rovers example in Figure 1, 2 Mars Rovers collaborate to explore rocks on the surface of Mars. The Rovers need to perform some sensor measurements on rocks, and, due to limited carriage capacity, can only carry 2 sensors at a time. Unused sensors are stored in base stations, and can be taken and returned to the base stations as needed.

The public facts in this problem are the sensors located in bases, and the current condition of the target rock. The public actions are taking and returning sensors to the base stations, putting collected rock samples at the base stations, and performing various examination actions on rocks, such as taking an image, mining a mineral, or collecting a sample. The sensors held by a rover and its position are private, and the private actions are movement actions.

3 Algorithmic Approaches

There are two major approaches to planning in CPPP (Torreño et al. 2017). The first approach begins by computing a public plan, which is known as a coordination scheme (Nissim, Brafman, and Domshlak 2010; Brafman and Domshlak 2013; Torreno, Onaindia, and Sapena 2014). Then, the agents independently extend the public plan into a complete plan by adding private actions. In this extension each agent attempts to achieve the preconditions of its own public actions in the public plan.

For example, in the DPP planner (Maliah, Shani, and Stern 2016b) the agents compute together a single agent projection of the CPPP problem that captures the dependencies between public actions. That is, which public actions facilitate the execution of other public actions of that agent. In our running example, such a dependency exists, e.g., for agent 1 between picking up a camera at base b_1 and taking a photo of the rock. These dependencies are computed using limited regression from the precondition of a public action to the effects of other public actions. Given this projection, one can compute a public plan using a standard classical planner. The projection is incomplete, and it is hence possible that the generated public plan cannot be extended to a complete plan, in which case DPP fails.

An alternative approach to computing a high level scheme is to compute a complete plan directly. This can be done by each agent running a distributed forward search algorithm over its own action space, informing other agents of advancements in the search process.

The first algorithm in this family is MAFS (Nissim and Brafman 2014) — a distributed algorithm in which every agent runs a best-first forward search to reach the goal. Each agent maintains an open list of states, and in every iteration each agent chooses a state in the open list to expand, generating all its children and adding them to the open list (avoiding duplicates). Whenever an agent expands a state that was generated by applying a public action, it also broadcasts this state to all other agents. An agent that receives a state adds it to the open list. For example, in Figure 1, when agent 1 puts down the camera at base b_2, it broadcasts this state to all other agents. Agent 2 can now use this state to pick up the camera and take a photo of the rock. To preserve privacy, the private part of a state is obfuscated when broadcasting it, e.g., by replacing the private facts with some index, such that only the broadcasting agent knows how to map this index to the corresponding private facts. Once the goal is reached, the agent achieving the goal informs all others, and the search process stops. MAFS can be extended in several ways.

Štolba and Komenda extended MAFS by applying two different open lists, one ordered by a local heuristic and the other by a global heuristic. Maliah et al. Maliah, Shani, and Brafman (2016) compute macros — sequences of private actions bounded by public actions — to expedite the local search process of the agent. For example, in Figure 1, once agent 1 has found the sequence of actions allowing it to get from base b_1 with the camera to the rock, it can save this sequence as a macro, allowing agent 1 to apply this macro in all future explored states where he wants to get from b_1 to the rock, expediting the search process.

4 Partial Disclosure of Private Dependencies

We now present the main contribution of this paper — reducing the amount of disclosed private dependencies and hence, the amount of disclosed information. In this paper, we discuss this in the context of the *DP Projection* method, which can be used either in a dedicated planner called the DP Planner or as a heuristic for MAFS-based solvers (Maliah, Shani,

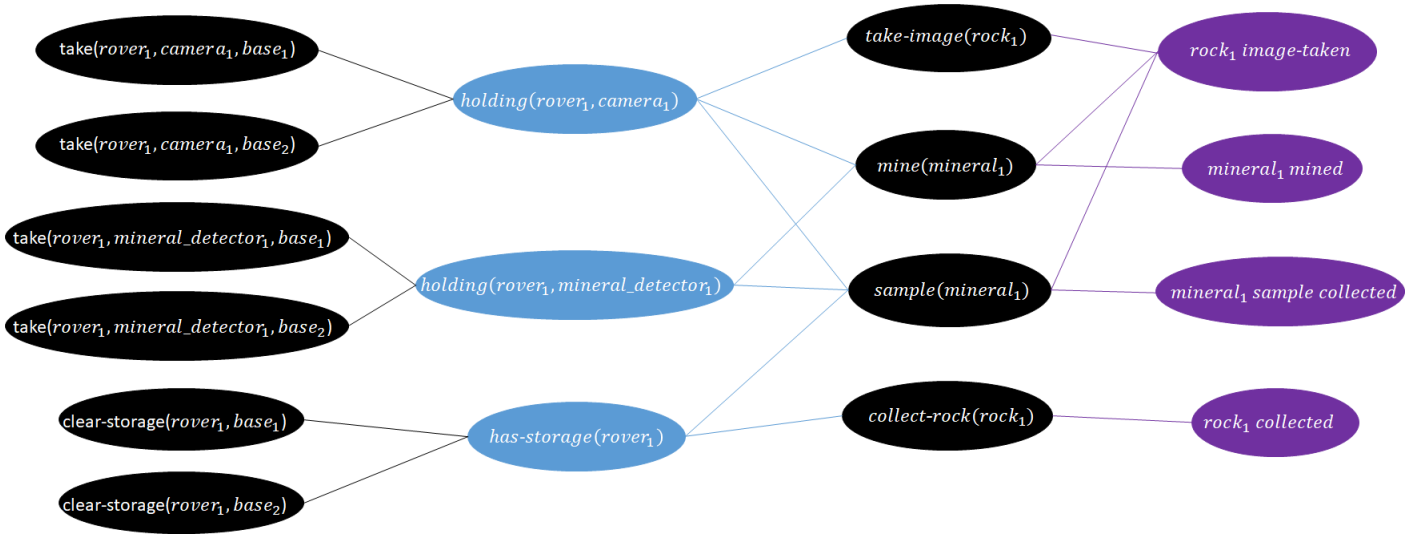


Figure 2: Local perspective of agent 1 private dependencies in the Rovers domain.

and Stern 2016a). We focus on the former application of the DP projection and only briefly discuss possible extensions to MAFS-based solvers.

For completeness and ease of exposition, we now describe a brief and slightly modified version of the DP projection. We say that a public action a *facilitates* the achievement of a private fact f, if (1) f is an effect of a, or (2) there exists a sequence of private actions $a_1, ..., a_k$ such that: f is an effect of a_k, each a_i takes as precondition an effect of some a_j s.t. $j < i$, and a_1 takes as precondition an effect of a.

Definition 1 (Private Dependency)**.** *An action a is said to have a private dependency if it has a private fact f as a precondition such that one of the following hold: (1) there is another public action a' that facilitates achieving f, (2) f is either true in the start state or can be achieved from it by only applying private actions.*

For example, in the *rovers* domain the action *take-image*$(rover_1, rock_1)$ has a private dependency because it has a private precondition *holding*$(rover_1, camera_1)$ that the public action *take*$(rover_1, camera_1, base_1)$ facilitates to achieve.

The agent jointly create a projection of the public problem, containing all the public facts, and a projected version of the public actions. For each public action a_i that requires a private precondition f_j, we create an artificial public fact f_j^i. The projected public version of a requires f_j^i as precondition. For each action a' of the agent that facilitates the achievement of the private f_j, the agent publishes f_j^i as an effect of a', thus publishing the private dependency of a on a', although the way that f_j is achieved remains obscured.

For a public action a_1 of an agent i we introduce a public artificial fact f_{a_1} into the projection, signifying that a_1 was executed. If a_1 facilitates the achievement of a private precondition of an action a_2 of agent i, then the projected a_2 will have f_{a_1} as precondition. Hence, the artificial predicates f_a capture the private dependencies between public actions.

The DP projection described by Maliah, Shani, and Stern (2016a) is created by having all agents compute and publish all their private dependencies. In this work, we limit to k the number of private dependencies of each agent is allowed to publish, where k is a parameter. Technically, each agent publishes all the artificial preconditions of all public actions, as well as k artificial effects of public actions. All public preconditions are published to avoid an over optimistic projection, where agents believe that they can execute public actions with no previous requirements.

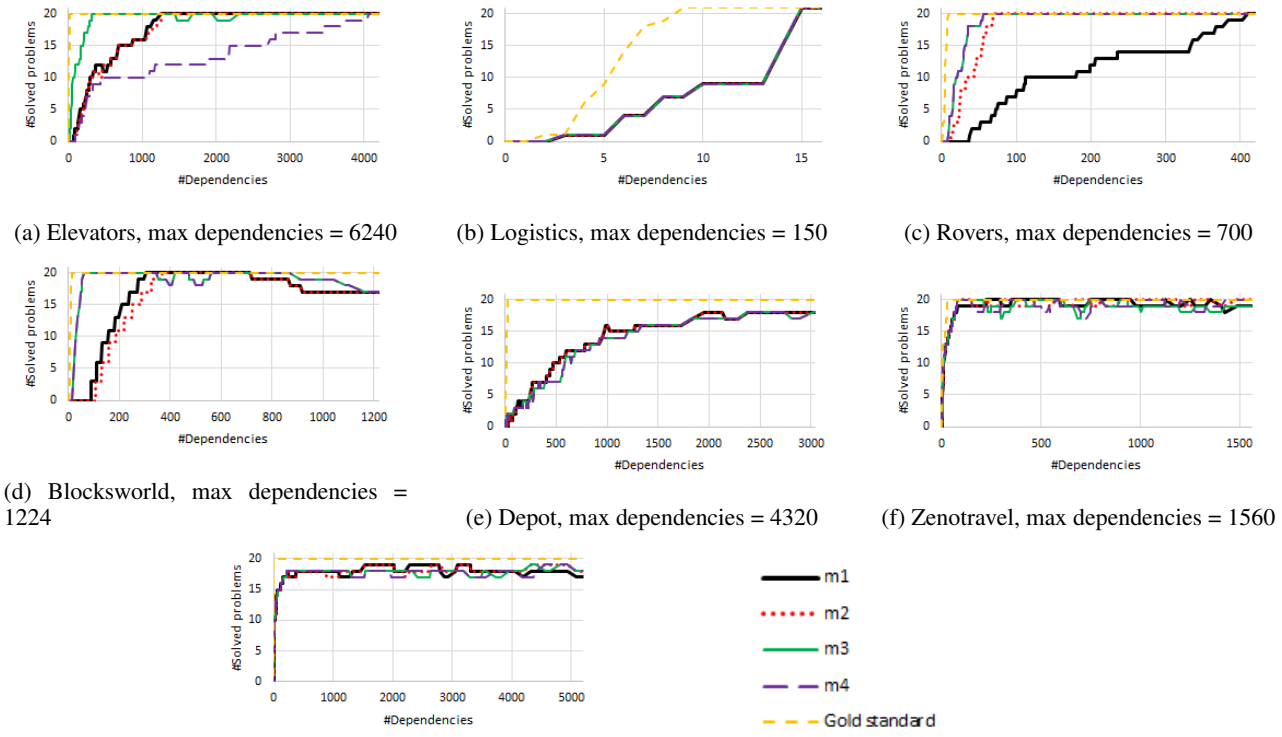
4.1 Theoretical Analysis

Before describing several heuristics for choosing which k private dependencies to share, we explore the theoretical implications of limiting the number of private dependencies that are being shared.

Let A and B be two DP projections. We denote by $A \subseteq B$ iff the set of private dependencies shared by A is a subset of the set of private dependencies shared by B. Let $plans(X)$ be the set of all plans that can be generated by a DP projection X. Let $P_{solve}(X)$ be the probability that a public plan chosen randomly from $plans(X)$ can be extended to a full plan (i.e., include also the private actions of the agents).

Theorem 1. *Given two DP projections A and B of the same problem, such that $A \subseteq B$, then: (1) $plans(A) \subseteq plans(B)$, and (2) $P_{solve}(A) \geq P_{solve}(B)$.*

Proof outline: Sharing a private dependency reveals that an additional artificial facts is achieved by some known public action. Artificial facts are only used as preconditions for some public actions. Thus, revealing private dependencies only facilitates performing additional actions, and thus more plans. The second part of Theorem 1 can be deduced from the fact that each link in the high level plan has a certain probability for extendability, and when increasing the amount of dependencies, you increase the links amount, bringing the probability to be lower than it was without that link.



(a) Elevators, max dependencies = 6240

(b) Logistics, max dependencies = 150

(c) Rovers, max dependencies = 700

(d) Blocksworld, max dependencies = 1224

(e) Depot, max dependencies = 4320

(f) Zenotravel, max dependencies = 1560

(g) Driverlog, max dependencies = 5800

(h) Legend.

Figure 3: Number of solved problems for each amount of published dependencies. Graphs truncated after all methods solve all problems. The maximal number of private dependencies in a problem in the chosen domain is shown for each domain

4.2 Ranking Published Dependencies

Above we discussed the implications of sharing k private dependencies, but we did not specify how to choose which k private dependencies to share. We now close this gap and suggest 4 different methods for selecting which dependencies to publish first. To better illustrate our methods, Figure 2 shows the local perspective of the private dependencies of agent 1 in the Rovers domain. Black nodes in the first and third column represent public actions, purple nodes in the fourth column represent public facts, and blue nodes in the second column denote artificial facts that capture private dependencies between public actions. The public actions on the first column generate the artificial facts while the public actions on the third column require them as preconditions, and generate the public facts. For ease of exposition, we label the artificial facts by the intuitive link that they represent, although in reality, of course, the artificial facts have no such meaningful names.

We take an iterative approach — all agents publish one artificial effect of one public action at each iteration. If the public projection cannot be solved, all agents publish a second artificial effect of a public action, and so forth. Hence, at each iteration an agent must decide on the next artificial effect to publish, given the effects it has published thus far.

Our first method, which we denote m_1, publishes artificial effects that are used as preconditions in as many public actions as possible. In the example in Figure 2, we can publish either the effect of *take*$(rover_1, camera_1, base_1)$, or the effect of *take*$(rover_1, camera_1, base_2)$, representing taking the camera from either base, as it supplies a precondition for 3 public actions (taking an image, mining a mineral, and collecting a stone sample). Let us assume that the effect of *take*$(rover_1, camera_1, base_1)$ was published first. To avoid that in the next round the effect of *take*$(rover_1, camera_1, base_2)$ will be chosen, providing the same preconditions, we subtract from the number of preconditions the amount of times this artificial effect was already published. Hence, at the next step, all unpublished effects have the same ranking.

The second method, which we denote m_2, publishes an effect that *enables the achievement* of as many public facts as possible. An effect enables the achievement of a public fact f if it is a precondition to an action that achieves f. In our illustration, this is when there is a path from a blue private fact to a purple public fact via a black node. Again, we subtract the number of times that an artificial fact was published as an effect to avoid repeatedly choosing the same effect. Hence, if we again select at the first iteration the effect of *take*$(rover_1, camera_1, base_1)$, at the second iteration we will select another artificial fact.

The third method, denoted m_3, attempts to maximize the amount of public actions that can be executed. That is, instead of publishing the artificial fact that provides a precondition for as many actions, we publish the artificial fact

Domain	Method	Min. cost	Max. cost	Min. dep. cost	Max. dep. cost	Improvement
Blocksworld	m1	47.70	95.90	75.00	48.10	34.92%
	m2	47.70	95.50	73.70	48.10	33.49%
	m3	43.50	101.00	74.90	43.90	37.69%
	m4	43.50	101.00	74.80	43.90	37.52%
Depot	m1	48.89	65.00	61.37	48.89	16.97%
	m2	48.89	64.63	61.37	48.89	16.97%
	m3	45.06	50.28	49.94	45.06	8.87%
	m4	45.06	50.28	49.94	45.06	8.87%
Driverlog	m1	37.58	67.32	63.63	38.63	28.12%
	m2	37.11	67.37	63.63	38.63	28.43%
	m3	38.00	67.05	63.32	38.63	27.37%
	m4	38.00	67.11	63.32	38.63	27.37%
Elevators	m1	65.15	87.10	68.00	80.60	4.60%
	m2	65.05	88.05	70.55	80.60	8.23%
	m3	64.60	84.45	68.85	80.60	8.37%
	m4	65.15	84.10	68.75	80.60	5.87%
Logistics	m1	59.65	66.10	65.45	60.25	8.52%
	m2	59.65	66.10	65.45	60.25	8.52%
	m3	59.75	66.20	65.45	60.25	8.47%
	m4	59.75	66.20	65.45	60.25	8.47%
Rovers	m1	72.85	80.10	78.35	75.30	6.59%
	m2	72.60	77.40	75.20	75.30	3.58%
	m3	72.90	77.75	75.85	75.30	3.76%
	m4	73.10	77.70	75.80	75.30	3.36%
Zenotravel	m1	68.25	73.20	69.90	71.20	2.97%
	m2	68.25	73.20	69.90	71.20	2.97%
	m3	68.00	73.45	69.90	71.20	2.73%
	m4	68.00	73.45	69.90	71.20	2.73%
Average	m1	57.15	76.39	68.81	60.43	14.67%
	m2	57.04	76.04	68.54	60.43	14.60%
	m3	55.97	74.31	66.89	59.28	13.89%
	m4	56.08	74.26	66.85	59.28	13.46%

Table 1: Averaged cost over all of the problems for each domain. The data in the "Improvement" column is the percentage of the minimal cost out of the first solved cost.

that enables as many public actions as possible. Here, in the first iteration, publishing the effect of taking the camera from either base, or the effect of clearing the storage on the rover, both enable immediately one public action (taking an image, and picking a sample into the storage bin, respectively). Assuming that we first select, again, *take*$(rover_1, camera_1, base_1)$, at the next iteration, taking the mineral detector from either base station or clearing the storage, both enable one additional action, and are hence tied.

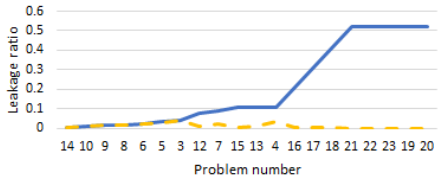
In this method we also need to balance between enabling new public actions that have not been available given the already published effects, and enabling new ways to apply already possible actions, that may result in better plans. As such, for each action a that is enabled by the published effect, we discount its score by $\frac{1}{c_a+1}$ where c_a is the number of times that a was enabled by previously published facts. Hence, at the first time that an action is enabled, it provides a score of 1, at the second time, a score of $\frac{1}{2}$, and so forth.

The last method, denoted m_4, is similar to m_3, but focuses not on the public actions, but on the public effects. That is, we publish an effect that enables achieving as many public facts as possible. Again, we discount the score of a public fact by $\frac{1}{c_f+1}$ where c_f is the number of times that public fact f was enabled by previously published facts. In this method, again taking the camera and clearing the storage are tied in the first iteration. In the second iteration, however, if we again select *take*$(rover_1, camera_1, base_1)$ first, taking the mineral detector has a better score, because it enables the mineral drilling action, that has 2 public effects, one of which was already obtained, and hence a score of $1\frac{1}{2}$.

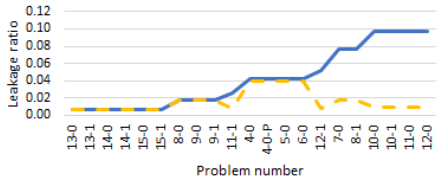
5 Empirical Analysis

We now evaluate our methods using benchmarks from the CPPP literature (Štolba, Komenda, and Kovacs 2015). For each problem in each benchmark domain, we ran the projection-based solver (Maliah, Shani, and Stern 2016a) with a growing number of revealed dependencies.

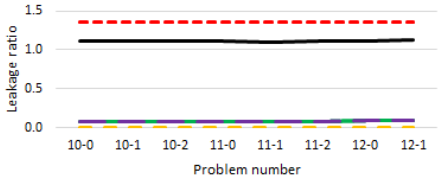
Figure 3 shows the number of problems that were solved on each domain given the number of revealed dependencies



(a) Zenotravel. Blue line represents all methods.



(b) Logistics. Blue line represents all methods.



(c) Blocksworld.

(d) Legend.

Figure 4: Privacy leakage in the various domains. Problems on the x-axis are sorted by increasing leakage.

by each agent. For each problem, we also computed a "Gold Standard" value, which is the number of private dependencies used by the plan that was generated when all dependencies are known. The gold standard serves as a baseline for comparison, indicating how many private dependencies are sufficient to find a solution.[1] In all domains, method m_3, which prioritize enabling additional public actions, performs the best. The rest of the methods vary in their performance. For example, on Rovers and Blocksworld, m_4, that prioritizes achieving additional public facts, performs the same as m_3 and better than the other methods, but on Elevators m_4 performs the worst. On Elevators, arguably the domain with the highest amount of required collaboration, differences between methods are most pronounced.

An interesting phenomena occurs, e.g., in Blocksworld, where some problems are solved when not all dependencies are available, but cannot be solved when more dependencies are published. This is because the projection method may produce a public plan that cannot be extended into a complete plans. Our methods were often able to publish dependencies that resulted in plans that could be extended. Later, additional dependencies confused the planner to choose plans that could not be extended. Zenotravel and Logistics are the easiest domains in the sense that all methods managed to obtain solutions for all problems after sharing only a small number of dependencies (note the scale in the x axis of the Logistics results). Most problems in the Driverlog domain were also easy in this sense, except for a few difficult problems which required more dependencies. On Depot, on the other hand, all methods perform much worse than the gold standard, leaving much room for improvement.

Next, consider the cost of the generated public plan. Table 1 shows for each domain the results of the following averaged factors: (1) Min. cost, (2) Max. cost, (3) Min. dep. cost – the solution cost for a projection with the least amount of dependencies revealed, and (4) Max. dep. cost – the solution cost for a projection with the maximal amount of dependencies revealed. In addition, the column "Improvement" shows the percentage of the minimal cost out of the first solved cost ($\frac{First\ solved\ cost - Minimal\ cost}{First\ solved\ cost} * 100\%$). We can see that on average (marked with green color at Table 1), the improvement is only about 14% for the different methods, which means that the first time the problem was solved gave us a pretty good solution (only 14% worse than the best solution that was found when we revealed more dependencies). However, the impact of sharing private dependencies on solution cost varies significantly per domain. For example, in Blocksworld the improvement is almost 40% while for Zenotravel it is always less than 3%. Also, note that in most cases there was almost no difference between the different heuristics (m1-4) in terms of solution cost.

Different methods require a different amount of published dependencies to solve the problem. While it is intuitive that revealing less dependencies preserves more privacy, this is not necessarily so. To measure the amount of privacy preserved by each algorithm, we use the privacy leakage tool[2] (Štolba, Tožička, and Komenda 2018) for measuring the amount of private information that is leaked by each method. Figure 4 shows the privacy leakage in 3 domains. For each problem in each domain, we take the first time that it was solved by each method, and generate the input for the privacy leakage tool. We measure the privacy leakage ratio from the perspective of agent 1 in each problem. The "Gold Standard" series's privacy leakage is calculated as if we only revealed those dependencies.

In Zenotravel and Logistics all methods publish the same number of dependencies before reaching the goal, although not necessarily the same dependencies. In these domains, the privacy leakage was identical for all methods, except for the gold standard, and is hence represented by a single line. In Blocksworld, however, the privacy leakage ratio of the different methods was not the same. As can be seen, there is a direct correlation between the amount of published dependencies required for computing a plan (Figure 3d), and the amount of leaked privacy (Figure 4c). This further sup-

[1] This does not mean, however, that it is not possible to find a solution if fewer private dependencies are revealed.

[2] http://github.com/stolba/privacy-analysis

ports our intuition that publishing less dependencies results in higher privacy. However, analysis of privacy leakage in CPPP is an ongoing research topic and it is difficult at this stage to draw general conclusions.

6 Conclusion and Future Work

In this work we suggest methods for publishing only a part of the private dependencies between public actions of agents, in order to reduce the amount of privacy leakage. We focus on the projection method that uses all private dependencies to compute a public plan, showing that in many cases a public plan can be computed with only a small portion of the dependencies, and that our heuristic methods rapidly find a good subset to share. We provide experiments over standard benchmark domains, comparing the coverage of our methods, as well as the amount of leaked privacy.

In the future we will expand our methods to MAFS, where an agent may publish only some public states that it reaches, and to other privacy preserving planning techniques as well. Another thing that we shall do in the future is finding the optimal set of dependencies that need to be revealed in order to solve each problem. This shall replace the gold standard that we have used now and will help us find better methods that will outperform the methods that were described in this paper. Future work will also include an analysis of the privacy leakage in the context of using the shared dependencies in the MAFS planner. Finally, it is also worthwhile to assign importance values to different dependencies, where each dependency has an intrinsic value that needs to be considered.

Acknowledgements

This work is partially funded by ISF grant # 210/17 to Roni Stern and by ISF grant # 1210/18 to Guy Shani.

References

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

[Brafman and Domshlak 2013] 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.

[Maliah, Shani, and Brafman 2016] Maliah, S.; Shani, G.; and Brafman, R. I. 2016. Online macro generation for privacy preserving planning. In *ICAPS*, 216–220.

[Maliah, Shani, and Stern 2016a] Maliah, S.; Shani, G.; and Stern, R. 2016a. Stronger privacy preserving projections for multi-agent planning. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, 221–229.

[Maliah, Shani, and Stern 2016b] Maliah, S.; Shani, G.; and Stern, R. 2016b. Stronger privacy preserving projections for multi-agent planning. In *ICAPS*, 221–229.

[Nissim and Brafman 2014] Nissim, R., and Brafman, R. I. 2014. Distributed heuristic forward search for multi-agent planning. *JAIR* 51:293–332.

[Nissim, Brafman, and Domshlak 2010] Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *AAMAS*, 1323–1330.

[Štolba and Komenda 2017] Štolba, M., and Komenda, A. 2017. The madla planner: Multi-agent planning by combination of distributed and local heuristic search. *Artificial Intelligence* 252:175–210.

[Štolba, Komenda, and Kovacs 2015] Štolba, M.; Komenda, A.; and Kovacs, D. L. 2015. Competition of distributed and multiagent planners (codmap). *The International Planning Competition (WIPC-15)* 24.

[Štolba, Tožička, and Komenda 2018] Štolba, M.; Tožička, J.; and Komenda, A. 2018. Quantifying privacy leakage in multi-agent planning. *TOIT* 18(3):28.

[Torreño et al. 2017] Torreño, A.; Onaindia, E.; Komenda, A.; and Štolba, M. 2017. Cooperative multi-agent planning: A survey. *ACM Comput. Surv.* 50(6).

[Torreno, Onaindia, and Sapena 2014] Torreno, A.; Onaindia, E.; and Sapena, O. 2014. FMAP: Distributed cooperative multi-agent planning. *Applied Intelligence* 41(2):606–626.

A Framework to prove Strong Privacy in Multi-Agent Planning

Patrick Caspari,[1] **Robert Mattmüller,**[2] **Tim Schulte**[3]
Department for Computer Science
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52
79110 Freiburg, Germany
[1]caspari@informatik.uni-freiburg.de,
[2]mattmuel@informatik.uni-freiburg.de,
[3]schultet@tf.uni-freiburg.de

Abstract

Privacy in multi-agent planning is an important and well-discussed problem. But it is difficult to prove whether an algorithm allows the deduction of private values. We introduce a framework that allows us to prove whether the messages transmitted during a Multi-Agent planning instance maintain strong privacy of the problem. It can be verified for an algorithm in general or for a specific execution of the algorithm on a planning task. We then use the framework to show that an A*-based secure-MAFS algorithm is in general not strongly privacy preserving.

Introduction

With the commercialization of automated machines and increasingly flexible and automated industry as well as the increase in computing power, the demand for algorithms that allow cooperative planning is high. But especially in the industry, where business secrets can be the foundation of success and their disclosure might mean the loss of many jobs, any cooperation can only be secure with tight guarantees for the privacy of their assets.

Several concepts of privacy exist and are used, depending on the degree of privacy that is necessary in the adopted domain. On the lower end of this range stands the notion of *weak privacy*, which is satisfied if an algorithm doesn't explicitly communicate private values to partners. While this is easy to verify, it doesn't account for the fact, that the transmitted information might still be used to deduce private information. On the other end, *strong privacy* requires that no agent can deduce the value or even the existence of another agent's private variables (Brafman 2015). While this definition is very strong, it is unclear how to verify whether an algorithm actually upholds it.

With the definition of PST-indistinguishability (Beimel and Brafman 2018), a verifyable lower bound for strong privacy was proposed. It states that all messages sent by an agent as well as their order must be uniquely defined by the public search tree. Other privacy definitions demand that the identity of the other agents is not revealed (Faltings, Léauté, and Petcu 2008) or that agents can't construct an upper bound on the number of objects of a type (Maliah, Shani, and Stern 2018).

We propose a framework that allows us to verify whether an execution of a multi-agent planning algorithm on a specific problem leaks private information. We then abstract from the specific problem and use the same framework to prove whether any exectution of a planning algorithm can leak private information. To further visualize this, we apply it to an A*-based secure-MAFS algorithm and show that it leaks private information.

Multi-Agent Planning

A multi-agent planning problem (Brafman and Domshlak 2008) is defined as a 4-tupel $\Pi = (P, \{A_i\}_{i=1}^N, I, G)$ where P is a set of propositions, A_i is a set of actions of an agent i, I the initial state and G a set of goal states. A part of the propositions $P^{\text{pub}} \subseteq P$ are shared among all agents. Other propositions $P^{\text{priv},i}$ are private to a single agent i. The existence of a private variable is only known to the respective agent and their value can only be affected by this agent's actions.

A *global* state s is defined as a valuation of all propositions $p \in P$. A state s^i of agent i is a valuation of all propositions $P^i = P^{\text{pub}} \cup P^{\text{priv},i}$ that affect this agent, that is all public propositions and those private to agent i. A *public* state s^{pub} only contains the valuations of the public variables. As a consequence, each public state describes a set of global states that differ in the private states of the individual agents.

We call an action *public* if the effect contains public variables. If the effect only contains private variables, the action is *private*. From each private action, we can generate a *public projection* a^{pub}, by stripping its precondition and effect of all private propositions.

During the execution of a planning algorithm X, the planner generates a *Planning Tree* $PT_X(\Pi)$. The planning tree is a directed graph which represents the paths and states that the algorithm expanded during the search. Each node of the graph corresponds to an expanded state; every edge represents the occurence of an action. Since an algorithm might expand states several times during its execution, sev-

eral nodes can refer to the same state.
We define a planning tree as a 6-tupel $PT = (N, E, i, V, D, d)$ where

- $N = \{n_1, \dots n_{|N|}\}$ is a set of nodes.
- $E = \{e_1, \dots e_{|E|}\}$ is a set of directed labeled edges. Each edge $e_k = (n^{\text{pre}}, n^{\text{post}}, \text{lab})$ consists of a predecessor node n^{pre}, a successor node n^{post} and a label lab corresponding to the action labels of Π.
- $i : N \to [0, |N|-1]$ is the indexing function that assigns a unique index to each node.
- $V : N \to 2^P$ is the valuation function that assigns a valuation over the propositions in Π to each node.
- D is a set of additional descriptions that may apply to a state. For our purposes, it models additional information that is transmitted during the application of a Multi-Agent Planning algorithm. This might for example be the path cost $g(s)$ or the heuristic function $h(s)$.
- $d : N \to D$ is the description function assigns the additional information to a node.

The planning tree is generated iteratively. At first, the planning tree only consists of the initial state. In each step, an new state is added to the tree by application of an action to one of the currently expanded states. The index of the nodes represents the order in which they were added to the planning tree. The initial state always carries the index 0.

We use an additional structure that we call the *Transmitted Tree* $\mathbf{T}$. It models the content of all messages transmitted between two agents. The transmitted tree has the same structure as the planning tree, with $\mathbf{T} = (N^T, E^T, i^T, V, D, d)$ and

$$N^T \subseteq N$$
$$E^T \subseteq T$$

The subsets N^T and E^T refer to the nodes and edges that were transmitted as messages by the algorithm. The indexing function i^T denotes the order in which the messages were received. Since not necessarily all expanded nodes are transmitted, the indices can vary from those of the planning tree.

In a purely public problem, if all expanded states and applied actions are transmitted, the transmitted tree is identical to the planning tree.

Strong Privacy

Brafman defined strong privacy as follows (Brafman 2015): *"A variable or a specific value of a variable is strongly private if the other agents cannot deduce its existence from the information available to them."*
Following this definition, it is not only the value of a private variable or its function whithin the problem that must stay unknown to the observer; even the fact that a private variable exists and has any effect on the execution of the algorithm must be obscured. In epistemic terms, we can rephrase this definition:

Definition 1 *A variable is strongly private if an observer can't distinguish between a world where this variable exists and affects the execution of the algorithm, and one where it doesn't exist.*

Initially, this may seem like a paradox - if the variable can't have a visible effect, what would be the point in using it? However, the definition doesn't say that the worlds are identical, but only that an observing agent can't tell which world it is in.

To illustrate this difference, consider three friends playing a card game. They play a single round, which player A wins. What the other players don't know is that player A is a skilled cheater. In a fair game, the chances of A winning would have been $\frac{1}{3}$; since they cheated, it is 1. Hence the *outcome* of the game differs from one where A plays fair. But from the perspective of B and C, the fact that they lost is in line with the assumption that the game was fair. So the other agents can't distinguish between a world where A cheated and one where they didn't.

The epistemic formulation implies another important property of strong privacy. The privacy of a value doesn't only depend on the algorithm but also on the observer. What is indistinguishable for one observer might be distinguishable for another.

For this, imagine that the card game from above is played with a non-standard deck of cards. Before the game, B looked through the cards and found out that two cards of each kind exist; player C didn't. Now A - still a cheater - has additional cards up their sleeve. They play out the third card of a kind, which is not supposed to exist. B's knowledge allows them to deduce that A cheated. From the perspective of C on the other hand the game still appears to be fair. So regarding C, the private property that A cheats is preserved; regarding B it isn't.

This shows that in order to discuss privacy in a multi-agent planning setting, we need to model the knowledge of an agent. In a general case it can be assumed that every agent knows what algorithm the others use and how that algorithm works. So we need a way to transform a planning algorithm into a knowledge model that we can check for consistency. Only then we can decide whether the execution of the algorithm and the exchanged messages are in accordance with the assumption that no private variables exist.

For simplicity, we regard the case of two agents: a sender S and a receiver R. The sending agent expands its private search tree and sends messages to R whenever the algorithm requires it. R acts as an honest, but curious agent, meaning that they apply the algorithm in the predefined manner, but try to deduce the private variables of S from the messages it receives.

We assume that the deduction occurs after the planning algorithm is complete, so that all messages are sent and the knowledge of R is static. R has the following knowledge, as illustrated in Figure 1:

- a set of actions
  $\mathbf{A}_S^{\text{pub}} : \{(a^{\text{pub}}, \text{pre}_a^{\text{pub}}, \text{post}_a^{\text{pub}}\}$
  that represents the public projections of the actions of S
- The *transmitted tree* $\mathbf{T}_S$ which models the messages exchanged between the agents. It contains the state descriptions and actions that were transmitted as part of the messages and are therefore known to be part of the planning

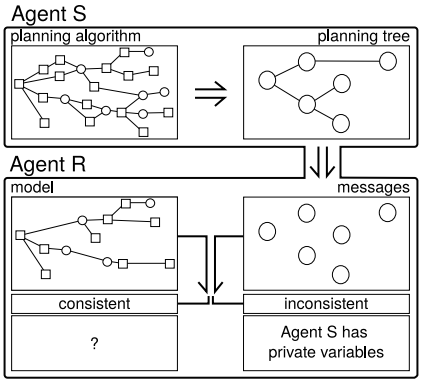


Figure 1: Schematic depiction of the deduction process.

tree.

- The *algorithm model* $\mathbf{M}_S$ which models the behavior of S's algorithm.

Specifically, the knowledge model contains no private variables of S. If R can construct a planning tree which is a possible product of the algorithm model and yields the messages that make up the transmitted tree, then S's privacy is preserved. However, if such a planning tree doesn't exist, R's knowledge isn't consistent with the actual execution of the algorithm. If we have a high trust in the algorithm model, the only possible reason for this inconsistency is that some private variable existed during the actual execution.

Definition 2 *We call the execution of a multi-agent planning algorithm on a planning problem* Π publicly self-sufficient*, if a planning tree exists, which doesn't contradict the agent's algorithm model model or the transmitted tree and contains only public variables.*

Equivalently we could say that a problem is publicly self-sufficient if the transmitted messages and the algorithm model are consistent with the assumption that no private variables exist.

Planning Logic

In order to prove whether tranmitted tree and algorihtm model are consistent, we need to represent both within a common framework, wich we call the *Planning Logic*. The planning logic is a set of propositions and logic rules that are able to model the execution of a multi-agent planning algorithm. It it based on the concept of planning as satisfiability (Kautz and Selman 1992). Since the valuations of the base propositions of the planning problem change from state to state, the logic needs a distinct set of base propositions for each state. Additionally, we need to model the actions with precondition and effect.

Definition 3 *Let* Π *be a multi-agent planning problem and* n_s *the number of states we want to consider. We call* $\mathcal{L}_\Pi$ *the* planning logic *induced by* Π *over the set of propositional atoms*

$$P_{\mathcal{L}_\Pi} = \{ \begin{array}{ll} p_{s_k} & \forall\, p \in P;\ k = 1, \ldots, n_s \\ a_{s_k,s_j} & \forall\, a \in A_S;\ k, j = 1, \ldots, n_s \end{array} \} \tag{1}$$

For a planning problem with $|P| = n_p$ propositions and $|A_S| = n_a$ actions and n_s reachable states, this yields a planning logic with $|P_{\mathcal{L}_\Pi}| = n_p \cdot n_s + n_a \cdot n_s^2$ propositional atoms.

Each state s_k in Π is defined by the valuation of the propositions P. We define each state in $\mathcal{L}_\Pi$ using a set of propositions p_{s_k}, with $p \in P$ and $k = 1, \ldots, n_s$. A positive literal p_{s_k} means equivalently that the proposition p evaluates as true in state s_k.

An action $a \in A_S$ is defined with a set of propositions a_{s_k,s_j} and two functions $\mathrm{pre}_a(s_k)$ and $\mathrm{apply}_a(s_k, s_j)$. The literal a_{s_k,s_j} denotes whether the action a was used in the transition from predecessor state s_k to successor state s_j. The function $\mathrm{pre}_a(s_k)$ returns the precondition of a in the propositions p_{s_k} of the provided predecessor state. Similarly, the function $\mathrm{apply}_a(s_k, s_j)$ returns the postcondition of a in the propositions p_{s_j} of the successor state and adds an additional term $p_{s_j} \leftrightarrow p_{s_k}$ for each base proposition p not mentioned in the postcondition of a.

During or after the execution of an algorithm, the generated planning tree can be represented as a valuation of all propositions of the planning logic. Let X be a planning algorithm and Π a planning problem. We call $\mathcal{L}(\cdot)$ the function that transfers a planning tree or a transmitted tree into a logic formular based on the planning logic.

Algorithm Models

In order to argue if the transmitted tree is compatible with the applied algorithm, we need a way to model the algorithm in the terms of the planning logic. To do this, we construct a set of axioms that define what planning trees this algorithm might produce.

Some axioms model properties that are common to all planning algorithms, for example that each node in the planning tree must be connected. Others may represent properties which are unique to the algorithm. An algorithm that searches the planning domain as a breadth-first search for example would need an axiom that defines that all reachable states are part of the planning tree.

In order to use the axioms we need to prove that they actually form a valid representation of the algorithm.

Definition 4 *Let* X *be a multi-agent planning algorithm and* A_X *a set of axioms.* A_X *is a* sound axiomisation *of* X *iff for any planning problem* Π

$$\forall PT \in \mathbf{PT}_X(\Pi)\ :\ \mathcal{L}(PT) \vDash A_X, \tag{2}$$

where $\mathbf{PT}_X(\Pi)$ *is the set of all planning trees the algorithm might generate from* X *on the problem* Π*.*

This means that all planning trees generated from Π with X

are consistent with the axioms in A_X. For a fully deterministic algorithm, $\mathbf{PT}_X(\Pi)$ contains only one entry. In a randomized algorithm, a repeated application of an algorithm X on a problem Π can lead to different planning trees. In that case, $\mathbf{PT}_X(\Pi)$ contains all possible planning trees that can be created that way.

The definition above guarantees that a set of axioms describes the algorithm. But it allows different axiomisations of the same algorithm. In fact, an empty set of axioms would be a sound axiomisation of any planning algorithm. This can be used to depict different levels of knowledge that an agent has. But in general, one would assume that an opposing agent has perfect knowledge about the applied algorithm. Accordingly, the knowledge model has to be a complete model of the applied algorithm.

Definition 5 *Let X be a multi-agent planning algorithm and A_X an axiomisation of this algorithm. Furthermore, let Π be a planning problem, and $\mathbf{PT}_X(\Pi)$ the set of all planning trees that the algorithm might generate from X.*
The axiomisation A_X of X is complete *iff for any planning problem Π*

$$\begin{aligned} &\forall PT_\Pi \in \mathbf{PT}_X(\Pi) \ : \ \mathcal{L}(PT_\Pi) \vDash A_X \text{ and} \\ &\forall PT'_\Pi \notin \mathbf{PT}_X(\Pi) \ : \ \mathcal{L}(PT'_\Pi) \nvDash A_X \\ &\text{with } P_{\mathcal{L}(PT'_\Pi)} = P_{\mathcal{L}(PT_\Pi)} \end{aligned} \tag{3}$$

where PT'_Π is another planning tree based on the problem Π.

In other words, it is impossible to construct a planning tree PT'_Π based on Π that fulfills the axioms A_X, but differs from those generated by X. This means that the axiomisation is a perfect representation of the planning algorithm. For a deterministic algorithm, a complete axiomisation is compatible with exactly one planning tree: the one that is generated by the planning algorithm on the public problem. For a randomized algorithm, the axioms are compatible with all planning trees in $\mathbf{PT}_X(\Pi)$.

Proving Public Self-Sufficiency

In Definition 2 we defined public self-sufficiency for one execution of an algorithm on a specific problem. In order to analyse whether an algorithm is publicly self-sufficent, we have to prove whether there are situations in which the algorithm can lead to a privacy leak.

Definition 6 *An algorithm X is* publicly self-sufficient *iff for any problem Π and any Execution E of the algorithm on Π*

$$\mathcal{L}(\mathbf{T}_X^E(\Pi)) \vDash A_X \tag{4}$$

where A_X is a complete axiomisation of X and $\mathbf{T}_X^E(\Pi)$ the transmitted tree generated during the execution E of the algorithm on problem Π.

Secure-MAFS

In the following, we will apply the discussed framework to an A*-based secure-MAFS algorithm (Brafman 2015). We will first construct an axiomisation of the A* algorithm and show that it is complete. Then, we will show that the algorithm is not publicly self-sufficient.

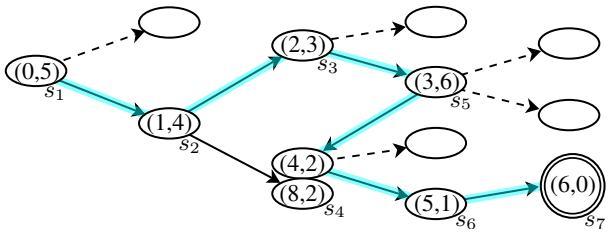


Figure 2: Possible search tree of a MAFS problem. The states are designated by their $(g(s), h(s))$ values. Note that s_4 is expanded twice with different paths costs. s_7 is a goal state.

Axiomisation

The A* algorithm doesn't expand the whole search tree. Instead, the algorithm keeps a list which contains all states that are whithin one action from an already expanded state. In every step it always expands the state s_k with the lowest expected path cost $f(s_k)$ from the list. If two states have the same expected path cost, a tie-breaking mechanism is applied. For our analysis we assume that the expected path cost function is chosen in a way that doesn't allow ties. This can be achieved by incorporating the tie-breaking mechanism into an expanded expected path cost function $\hat{f}(\cdot)$.

We introduced a set of descriptors D to model the additional information transmitted during the execution of a MAP algorithm. In the case of secure-MAFS, this would be the codomains $\mathbb{D}_g$ and $\mathbb{D}_h$ of the path cost function and the heuristic function.

$$\begin{aligned} g \ &: \ S \to \mathbb{D}_g \\ h \ &: \ S \to \mathbb{D}_h \\ D \ &:= \mathbb{D}_g \times \mathbb{D}_h \end{aligned} \tag{5}$$

Since secure-MAFS is an informed search algorithm, and since we want to assume full knowledge of the observer, we have to assume that the functions $g(\cdot)$ and $h(\cdot)$ are known to the observer.

An axiomisation A_{A^*} of the A* algorithm can be defined as follows:

1. *Action definition* An action a can only be applied in a state s if the action's precondition is satisfied in s. Similarly, the valuation of the successor state s' must satisfy the postcondition of a. All propositions not mentioned in the postconditions must stay unchanged between s and s'.

$$a_{s,s'} \to \mathrm{pre}_a(s) \wedge \mathrm{apply}_a(s, s') \tag{6}$$

   The functions $\mathrm{pre}_a(s)$ and $\mathrm{apply}_a(s, s')$ are as defined earlier. This axiom follows directly definition of an action and therefore has to apply to all possible planning trees and planning algorithms.

2. *Reachability* Every state except for the initial state is added to the open_list as successor of a previously expanded state. As a consequence, the first time a state is

expanded, its predecessor state must have a lower index.

$$\forall s_k \text{ with } k > 1 \; \exists a, s_j \text{ with } j < k \; : \; a_{s_j,s_k} \tag{7}$$

Since the initial state is the state with the smallest index, and the only one that doesn't need a predecessor with smaller index, this axiom impies that all states are reachable from the initial state s_1.

3. *Singleton States* A state is only reexpanded if a shorter path to it is found. So if two states have increasing expected path costs f, they have to differ in at least one proposition.

$$\begin{aligned} &\forall s_j, s_k, j < k \text{ with } f(s_j) < f(s_k) \; : \\ &\quad \exists p : (p_{s_k} \wedge \neg p_{s_j}) \vee (\neg p_{s_k} \wedge p_{s_j}) \end{aligned} \tag{8}$$

4. *No Backward Edges* Before a state s_k is expanded, it has to be added to the open_list. This only happens if an action from a previously expanded state leads to s_k. So its predecessor must have been expanded before and thus have a lower index. As we defined above, all nodes in the planning tree are expansions of a state.

$$\forall s_k, s_j, k \leq j \; : \; \neg a_{s_j,s_k} \tag{9}$$

5. *Expansion Order* Before a state s_k is expanded, a number of possible states is considered reachable (in the open_list). They can be seen as the successor states of all currently applicable actions. Out of these states, s_k must have the lowest f-value.

$$\begin{aligned} &\forall s_k, s_j, \text{ with } j < k \; : \\ &\forall a \text{ with } \mathrm{pre}_a(s_j) \text{ and } f(\mathrm{apply}_a(s_j)) < f(s_k) \; : \\ &\exists s_l \text{ with } l < k \; : \; a_{s_j,s_l} \end{aligned} \tag{10}$$

The formular reads as follows: all states reachable from s_j whose expected path cost is lower than that of s_k must be expanded before s_k.

6. *Shortest Path* If two actions are both applicable in different states, and both would expand a new state with identical valuations, only the one with the lower expected path cost is applied.

$$\begin{aligned} &\forall s_j, s_k, s_l \text{ with } j, k < l \text{ and} \\ &a^A, a^B \text{ with } \mathrm{pre}_{a^A}(s_j) \wedge \mathrm{pre}_{a^B}(s_k) \text{ and} \\ &\mathrm{apply}_{a^A}(s_j) = \mathrm{apply}_{a^B}(s_k) = V(s_l) \text{ and} \\ &f\big(\mathrm{apply}_{a^A}(s_j)\big) < f\big(\mathrm{apply}_{a^B}(s_k)\big) \; : \\ &\neg a^B_{s_k,s_l} \end{aligned} \tag{11}$$

This also implies that each node has only one predecessor with lower index. In combination with axiom 4, which excludes any predecessor with higher index, each node can only have exactly one preceding action.

7. *Path Cost and Heuristic Value* The transmitted path- and heuristic cost $g_s,\ h_s$ that are transmitted in the messages must align with the values computed by the observer.

$$\forall s_k \; : \; h(s_k) = h_{s_k}, \;\; g(s_k) = g_{s_k} \tag{12}$$

While the last axiom may seem obvious, it illustrates an important rule concerning the properties of a privacy preserving algorithm: A MAP algorithm can only preserve privacy if the functions $g(\cdot)$ and $h(\cdot)$ only depend on public variables.

We won't formally prove that the axiomisation is sound according to definition 4. In the following, we will assume that definition 4 holds and based on this assumption prove that A_{A^*} is a complete axiomisation according to definition 5.

Consider the set $\mathbf{PT}_{\mathrm{ax}}(\Pi)$ of all planning trees based on problem Π which satisfy the axioms A_{A^*}. We assume that the axiomisation A_{A^*} is sound, so $\mathbf{PT}_{\mathrm{ax}}(\Pi)$ must contain $\mathbf{PT}_{\mathrm{A}^*}(\Pi)$. Since secure-MAFS is a deterministic algorithm, the set $\mathbf{PT}_{\mathrm{A}^*}(\Pi)$ contains only one entry PT_Π.

$$\mathbf{PT}_{\mathrm{A}^*}(\Pi) = \{PT_\Pi\} \subseteq \mathbf{PT}_{\mathrm{ax}}(\Pi)$$

If we can prove that $\mathbf{PT}_{\mathrm{ax}}(\Pi)$ is also singular, so the axiomisation is only consistent with exactly one planning tree PT_Π for any problem Π, then $\mathbf{PT}_{\mathrm{A}^*}(\Pi) = \mathbf{PT}_{\mathrm{ax}}(\Pi)$ and the axiomisation is complete. We will prove this by induction over the nodes of the planning tree.

Lemma 1: Uniqueness
A planning tree PT_Π that fulfills the axiomisation A_{A^*} is uniquely defined.
Induction start: s_1
The valuation of the initial state s_1 is defined by the problem Π.

Induction hypothesis:
The partial tree PT_Π^{k-1} defined by the states $s_1, \ldots, s_{k-1}$ and all applied actions between them is uniquely defined.

Induction step: s_k
Let $S_k^{\mathrm{pre}} = \{s_1, \ldots, s_{k-1}\}$ be the set of all states in PT_Π^{k-1}. Per induction hypothesis, all those states and their valuations are uniquely defined.

Axiom 2 defines that s_k must be the successor of another state s_j via an action a^{s_k}. Due to axiom 4, the preceding state must have a lower index, so we know that $s_j \in S_k^{\mathrm{pre}}$. Consequently, s_k must be reachable from a state in S_k^{pre} by application of one action. We call $S_k^{\mathrm{pot}} = \bar{s}_1, \ldots, \bar{s}_n$ the set of potential states reachable from S_k^{pre} in this manner. We sort the potential states by f-value, with $f(\bar{s}_1) < \cdots < f(\bar{s}_n)$. This is possible because we defined $f(\cdot)$ to be an injective function by including the tie-breaking function in it.

Consider a state $\bar{s}_i \in S_k^{\mathrm{pre}}$ as candidate for s_k. Axiom 5 demands that all reachable states with lower f-value - in this case the states $\bar{s}_1, \ldots, \bar{s}_{i-1}$ - have a lower index. All states with an index lower than k are in the set S_k^{pre}. It follows that $\{\bar{s}_1, \ldots, \bar{s}_{i-1}\} \subseteq S_k^{\mathrm{pre}}$.

All states in S_k^{pre} are already part of the partial planning tree PT_Π^{k-1}. Consequently, s_k must be the first entry among S_k^{pot} (as sorted by f-value) that is not already part of PT_Π^{k-1}. This uniquely defines both s_k and its preceding action a^{s_k}.

Axiom 6 implies that each state has only one preceding action. Therefore the new partial tree PT_Π^k generated by

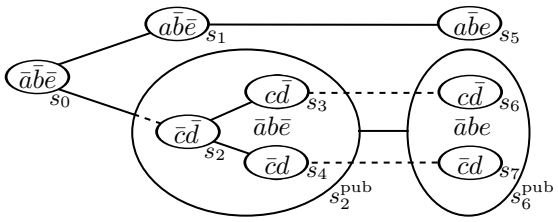


Figure 3: Public and private search tree as processed by secure-MAFS. The propositions a, b and e are public, c and d are private.

adding s_k and a^{s_k} to PT_{Π}^{k-1} doesn't allow the addition of any further actions between the states. Therefore, all states and actions of PT_{Π}^{k} are uniquely defined. □

It follows from the lemma that $\mathbf{PT}_{\mathrm{A}^*}(\Pi) = \mathbf{PT}_{\mathrm{ax}}(\Pi)$. Therefore, A_{A^*} is a complete axiomisation of the A^* algorithm.

Public Self-Sufficiency

The secure-MAFS algorithm only sends valuations of states in the messages, not which actions were used to get to these states. So at the end of the execution, the transmitted graph consists of an unconnected set of states.

Secure-MAFS is an extension of MAFS that was developed to give better privacy guarantees. Instead of just keeping a list of open states it maintains a list of the public projections of these states. Each of these open public states has two lists attached: a list of open private states that have this same public projection and a list of public states s' which are successors of any state in the first list. Figure 3 illustrates this: The private states s_3, s_4, s_5 and s_7, s_8 are merged to two single public states t_1 and t_2. As t_2 is the successor of t_1, the open list of this problem would look as follows:

public states:	$\bar{a}\bar{b}\bar{e}$	$a\bar{b}\bar{e}$	$\bar{a}b\bar{e}$	$a\bar{b}e$	$\bar{a}be$
private states:	$\bar{c}\bar{d}$	$\bar{c}\bar{d}$	$\bar{c}\bar{d}$ $c\bar{d}$ $\bar{c}d$	$\bar{c}\bar{d}$	$c\bar{d}$ $\bar{c}d$
successors:	$a\bar{b}\bar{e}$ $\bar{a}b\bar{e}$	$a\bar{b}e$	$\bar{a}be$	-	-

Instead of broadcasting every state to the other agents, the secure-MAFS algorithm adds new private states to the list and only sends a new message when a new public state is added to the open list.

As discussed earlier, an informed search algorithm can only be privacy preserving if the path cost $g(\cdot)$ and the heuristic $h(\cdot)$ only depend on public properties; therefore we will assume this for this analysis.

To prove or disprove strong privacy, we will go through the axioms and show whether they can stand in conflict to the transmitted graph. In order to keep the clarity of the prove, we will only consider the interactions of each axiom with the previous axioms. We call s_k a global state with public and private propositions, s_k^{priv} a purely private state that is not transmitted during the application of MAFS and $s_k^{\triangleright}$ the public projection of s_k. Equivalently, $a^{\triangleright}$ is the public projection of an action a that is transmitted to the other agents.

1. *Action Definition*

$$a^{\triangleright}_{s_j^{\triangleright}, s_k^{\triangleright}} \rightarrow \mathrm{pre}_{a^{\triangleright}}(s^{\triangleright}) \wedge \mathrm{apply}_{a^{\triangleright}}(s_k^{\triangleright}, s_j^{\triangleright})$$

Since the transmitted graph initially contains no actions, this axiom cannot stand in contrast do it.

2. *Reachability*

$$\forall s_k^{\triangleright} \text{ with } k > 1 \; \exists a^{\triangleright}, s_j^{\triangleright} \text{ with } j < k \; : \; a^{\triangleright}_{s_j^{\triangleright}, s_k^{\triangleright}} \qquad (13)$$

Every state except the initial state has to be the successor of another state with lower index. In combination with axiom 1 this means that the public projection $s_k^{\triangleright}$ of a state s_k needs a public predecessor $s_j^{\triangleright}$, $j < k$ and a public action $a^{\triangleright}$ such that $\mathrm{pre}_a^{\triangleright}(s_j^{\triangleright})$ and $\mathrm{apply}_{a^{\triangleright}}(s_j^{\triangleright}, s_k^{\triangleright})$ are true.
The last action that leads to a transmitted state is always a public action. So the public projection of the postconditions of that action are satisfied in s_k.

$$\forall s_k \; \exists a \; : \; \mathrm{post}_{a^{\triangleright}}(s_k^{\triangleright})$$

Before every public action can come an arbitrary number of private actions. By definition, a private action only has private variables in its effect. Accordingly, between two transmitted states s_j and s_k are a number of untransmitted private states $(\bar{s}_1^{\text{priv}}, \ldots, \bar{s}_n^{\text{priv}})$ which all have the same public projection as the public predecessor s_j. Since the public projection of the public predecessor is the same as the projection of the actual predecessor $s_j^{\triangleright} = \bar{s}_n^{\text{priv}\triangleright}$, and the public projection of the precondition $\mathrm{pre}_a^{\triangleright}$ is applicable in $\bar{s}_n^{\text{priv}}$, it must also be applicable in s_j.

$$\forall s_k \; \exists a, s_j \; : \; \mathrm{pre}_{a^{\triangleright}}(s_j^{\triangleright}) \wedge \mathrm{post}_{a^{\triangleright}}(s_k^{\triangleright})$$

Since the only thing that affects the public projection of the states between s_j and s_k is the public action a, we can conlude that all public propositions not mentioned in $\mathrm{post}_{a^{\triangleright}}$ stay the same.

$$\forall s_k \; \exists a, s_j \; : \; \mathrm{pre}_{a^{\triangleright}}(s_j^{\triangleright}) \wedge \mathrm{apply}_{a^{\triangleright}}(s_j^{\triangleright}, s_k^{\triangleright})$$

In conclusion, the *single root state* axiom is also satisfied in any execution of secure-MAFS.

3. *Singleton States*

$$\forall s_k, s_j \text{ with } k < j, \; f(s_k) \leq f(s_j) :$$
$$\exists p \in P^{\text{pub}} : (p_{s_k} \wedge \neg p_{s_j}) \vee (\neg p_{s_k} \wedge p_{s_j})$$

Whenever secure-MAFS extends a state, it checks whether that another state with the same public projection was already transmitted. If so, the state is not transmitted to the other agents. This guarantees that no public state is transmitted twice and therefore that this axiom always holds.

4. *No Backward Actions*

$$\forall s_k, s_j, k \leq j \; : \; \neg a_{s_j, s_k} \qquad (14)$$

All actions implied by axiom 2 lead from a previously expanded state to one that is expanded later on. Therefore, this axiom is always satisfied.

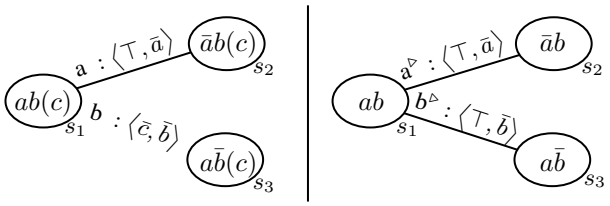


Figure 4: Left: global states; right: public projection of the same states. In the public projection, the state $(a\bar{b})$ appears to be reachable, while in fact it is not.

5. *Expansion Order*

$$\begin{aligned}&\forall \text{ public } s_j, s_k, \text{ with } j<k \ : \\ &\forall a \text{ with } \mathrm{pre}_{a^\triangleright}(s_j^\triangleright) \text{ and } f(\mathrm{apply}_{a^\triangleright}(s_j)) < f(s_k) \ : \\ &\exists \text{ public } s_l \text{ with } l<k \ : \ a_{s_j,s_l}\end{aligned} \tag{15}$$

Consider a situation as shown in figure 4. While $s_3^\triangleright$ appears to be reachable in the public projection of the problem, it is in fact not because the private variable c prohibits it.
If $f(s_3) < f(s_2)$, the axiom requires $s_3^\triangleright$ to be expanded before $s_2^\triangleright$. But since s_3 can't be expanded, this situation leads to a violation of axiom 5.

6. *Shortest Path*

$$\begin{aligned}&\forall s_j, s_k, s_l \text{ with } j,k<l \text{ and} \\ &a^A, a^B \text{ with } \mathrm{pre}_{a^{A\triangleright}}(s_j^\triangleright) \wedge \mathrm{pre}_{a^{B\triangleright}}(s_k^\triangleright) \text{ and} \\ &\mathrm{apply}_{a^{A\triangleright}}(s_j^\triangleright) = \mathrm{apply}_{a^{B\triangleright}}(s_k^\triangleright) = V(s_l^\triangleright) \text{ and} \\ &f\big(\mathrm{apply}_{a^{A\triangleright}}(s_j^\triangleright)\big) < f\big(\mathrm{apply}_{a^{B\triangleright}}(s_k^\triangleright)\big) \ : \\ &\neg a^B_{s_k,s_l}\end{aligned} \tag{16}$$

The same reasoning as for axiom 5 applies. If the public projection $a^{A\triangleright}$ is applicable, but the actual action a^A is not, then $a^B_{s_k,s_l}$ is the only path that connects s_l to the rest of the planning tree. Consequently, axiom 6 is not guaranteed to hold.

7. *Path Cost and Heuristic Value*

$$\forall s_k \ : \ h\big(V(s_k)\big) = h_{s_k}, \ \ g\big(V(s_k)\big) = g_{s_k} \tag{17}$$

Since we assumed that the path cost and heuristic value depend only on public propositions, this is by construction always fulfilled.

Axiom 5 and 6 aren't guaranteed to hold. This means that secure-MAFS is not a public self-sufficient algorithm.

Logistics Example

We will illustrate this with an example from the logistics domain. A truck needs to deliver two packages `a` and `b` from their respective locations `A` and `B` to a third location `X`. The problem has the following public propositions

`A` : truck is at location `A`
`B` : truck is at location `B`
`X` : truck is at location `X`
`a` : package `a` is loaded in the truck

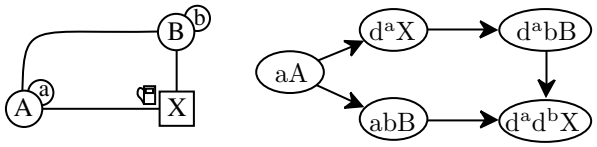


Figure 5: Logistics example. Left: the different locations and paths; right: public search tree.

`b` : packabge `b` is loaded in the truck
$\mathtt{d}^\mathtt{a}$: package `a` is delivered
$\mathtt{d}^\mathtt{b}$: package `b` is delivered

Figure 5 shows the paths between the locations and the corresponding public search tree of the truck. To keep the example readable we omit negative propositions in all state descriptions. The truck has the following public actions:

$$\begin{aligned}&\mathtt{drive}(\mathtt{L}_1,\mathtt{L}_2) &&: \langle \mathtt{L}_1 \mid \bar{\mathtt{L}}_1\mathtt{L}_2\rangle \\ &\mathtt{pick\_up}(\mathtt{L},\mathtt{p}) &&: \langle \mathtt{L}\bar{\mathtt{p}}\bar{\mathtt{d}}^\mathtt{p} \mid \mathtt{p}\rangle \\ &\mathtt{deliver}(\mathtt{p}) &&: \langle \mathtt{X}\mathtt{p} \mid \bar{\mathtt{p}}\mathtt{d}^\mathtt{p}\rangle\end{aligned}$$

The placeholders $\mathtt{L}_1, \mathtt{L}_2, \mathtt{L} \in \{\mathtt{A}, \mathtt{B}\}$ describe locations and $\mathtt{p} \in \{\mathtt{a}, \mathtt{b}\}$ describes a package. The planning logic $\mathcal{L}_{\Pi^\triangleright}$ of the public problem $\Pi^\triangleright$ consists of the propositions

$$\begin{aligned}P_{\mathcal{L}_{\Pi^\triangleright}} = \{ \\ A_{s_k}, B_{s_k}, X_{s_k}, a_{s_k}, b_{s_k}, d^a_{s_k}, d^b_{s_k}, \\ dr^{L_1L_2}_{s_j s_k}, p^{Lp}_{s_j s_k}, d^p_{s_j s_k}, \\ k,j = 1,\ldots,|S|;\ p = a,b;\ L, L_1, L_2 = A, B, X\}\end{aligned} \tag{18}$$

The public search tree as shown in figure 5 can be transcribed as the following logic formular

$$a_{s_1} A_{s_1} \cdot d^a_{s_2} X_{s_2} \cdot a_{s_3} b_{s_3} B_{s_3} \cdot d^a_{s_4} b_{s_4} B_{s_4} \cdot d^a_{s_5} d^b_{s_5} X_{s_5} \tag{19}$$

again we omit all negative propositions for better readability.

The amount of fuel in the tank of the truck is considered private and has the following possible states:

$\mathtt{t}^\mathtt{f}$: tank is full
$\mathtt{t}^\mathtt{h}$: tank is half full
$\mathtt{t}^\mathtt{e}$: tank is empty

As it happens, driving between locations `A` and `B` uses up a full tank, while between `A` and `X` or `B` and `X` only uses half a tank. The truck can only refuel at location `X`. The private actions of the truck are as follows:

$$\begin{aligned}&\mathtt{drive\_l}(\mathtt{L}_1,\mathtt{L}_2) &&: \langle \mathtt{L}_1\mathtt{t}^\mathtt{f} \mid \bar{\mathtt{L}}_1\mathtt{L}_2\mathtt{t}^\mathtt{e}\rangle \\ &\mathtt{drive\_s}(\mathtt{L}_1,\mathtt{L}_2) &&: \langle \mathtt{L}_1\bar{\mathtt{t}}^\mathtt{e} \mid \bar{\mathtt{L}}_1\mathtt{L}_2(\mathtt{t}^\mathtt{f} \triangleright \mathtt{t}^\mathtt{h})(\mathtt{t}^\mathtt{h} \triangleright \mathtt{t}^\mathtt{e})\rangle \\ &\mathtt{refuel} &&: \langle \mathtt{X} \mid \mathtt{t}^\mathtt{f}\rangle\end{aligned}$$

The action `drive_l` denotes the long road between `A` and `B`, `drive_s` denotes the short road from or to location `X`.

We apply the secure-MAFS algorithm to this problem. We define the path cost function $g(s)$ as the number of `drive` actions necessary to reach a state. As heuristic function $h(s)$ we use the optimal heuristic h^*. The initial state of the truck is ($\mathtt{Aat}_\mathtt{h}$): The truck is at location `A` with only package `a` and a half empty tank.

Figure 5 shows two different paths to the goal state. The upper path via state ($\mathtt{d}^\mathtt{a}\mathtt{X}$) has a path cost of 3, the lower path

via the state (abB) a cost of 2. Since we assume a perfect heuristic h^*, those are also the f-values of the mentioned public states. Expanding the cheaper, lower path would require the agent to first expand the action drive_l(A, B). Because the agent has only half a tank of gas, this action is not applicable in the initial state. The agent can only expand the more expensive upper path and transmit the corresponding message (d^aX). This violates axiom 5, because the state (abB) has a lower f-value. This allows the observer to deduce that some private variable must exist that doesn't allow the expansion of the lower path.

Conclusions

Optimality vs. Privacy

The problem of secure-MAFS we presented is a general one: If an observer has perfect knowledge of the applied algorithm, it can generate the public search tree of the sender and find the optimal path in it. If an action in that path is not applicable because of a private variable, the transmitted optimal path differs from the expected one. This allows the observer to deduce that private variables exist and affect the execution of the algorithm.

It was shown before that a multi-agent planning algorithm can't simultaneously be strongly private, optimal and efficient (Tožička, Štolba, and Komenda 2017). We can use this knowledge and trade in some optimality or efficiency for improved privacy. A private, but non-optimal variant of secure-MAFS would expand the search tree using a random walk. Instead of expanding the node with the lowest f-value, it would choose a random node from the open list. The axiomisation A_{rMAFS} would consist of axioms 1-4 and 7 from above; the axioms *Expansion Order* and *Shortest Path* would become obsolete. The axioms of rMAFS describe a set of plannig trees, all of which could be generated by the algorithm. The observer can't distinguish whether an action is omitted because it isn't applicable or just because a different action was randomly chosen.

Comparison to β-indistinguishability

We will compare our privacy definition to *PST-indistinguishability* (Beimel and Brafman 2018). *PST-indistinguishability* claims that the messages R receives during the application of a MAP algorithm are uniquely defined by the public search tree of the problem and the initial state of R.
In a more general setting, *β-indistinguishability* considers not the public search tree but the output of a leakage function $\beta(\Pi)$. The exact definition of β-indistinguishability as given in (Beimel and Brafman 2018) is as follows:

Definition 7 *Let $\beta : \{0,1\}^* \to \{0,1\}^*$ be a (leakage) function. We say that a deterministic algorithm is β-indistinguishable if there exists a simulator* Sim *such that for every set T of agents and for every input $x = (x_1, ..., x_n)$ the view of T is the same as the output of the simulator that is given $(x_i)_{i\in T}$ and $\beta(x)$, i.e.,* $\mathrm{Sim}(T, (x_i)_{i\in T}, \beta(x)) = \mathrm{view}_T(x)$.

Instead of working with the public search tree, we define a different construction that we will call the *transmitted graph*. The transmitted graph consists of the states and actions that are transmitted by the algorithm. Since a privacy preserving algorithm only transmits the public projections of actions and states, the transmitted graph consists of a subset of the states and actions in the public search tree.
We call an algorithm *public self-sufficient* if the transmitted graph is consistent with the model a foreign agent has of the algorithm.

If a receiving agent R has perfect knowledge of the sender i.e. if the model $\mathbf{M}_S$ is an exact representation of the applied algorithm including possible heuristics, we can regard the model as a simulator Sim from definition 7. The leakage function β is then only given by the transmitted messages. An unsatisfiable model and transmitted graph is then equivalent with the simulator being inconsistent with the transmitted messages.

Further Work

Further work is needed to evaluate the implications of public self-sufficiency. New algorithms are needed that comply to the stricter rules of public self-sufficiency. One interesting approach might be to check before transmitting a message, whether this message might compromise privacy and to adjust it accordingly or expand a different path. Another promising path is that of randomized algorithms. Especially in a larger problem, they might be vunerable to a stochastic approach. It is also important dicuss further, when we want to trade optimality for privacy, and in what domains more relaxed concepts of privacy are sufficent.

References

Beimel, A., and Brafman, R. 2018. Privacy preserving multi-agent planning with provable guarantees. *ArXiv*.

Brafman, R., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. *Amerian Association for Artificial Intelligence*.

Brafman, R. 2015. A privacy preserving algorithm for multi-agent planning and search. *International Joint Conference on Artificial Intelligence*.

Faltings, B.; Léauté, T.; and Petcu, A. 2008. Privacy guarantees through distributed constraint satisfaction. 350–358.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. 359–363.

Maliah, S.; Shani, G.; and Stern, R. 2018. Action dependencies in privacy-preserving multi-agent planning. *Autonomous Agents and Multi-Agent Systems* 32(6):779–821.

Tožička, J.; Štolba, M.; and Komenda, A. 2017. The limits of strong privacy preserving multi-agent planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Decentralized Acting and Planning Using Hierarchical Operational Models

Ruoxi Li, Sunandita Patra, Dana Nau
Dept. of Computer Science, and Institute for Systems Research
University of Maryland, College Park, MD 20742, USA
rli12314@cs.umd.edu, patras@umd.edu, nau@cs.umd.edu

Abstract

We describe Dec-RAE-UPOM, a system for decentralized multi-agent acting and planning in environments that are partially observable, nondeterministic, and dynamically changing. Dec-RAE-UPOM includes an acting component, Dec-RAE, and a planning component, UPOM. The acting component is similar to the RAE acting engine (Ghallab, Nau, and Traverso 2016), but incorporates changes that enable it to be used by autonomous agents working independently in a collaborative setting. Each agent runs a local copy of Dec-RAE and has its own set of hierarchical operational models that specify various ways to accomplish its designated tasks. Agents can communicate with each other to exchange information about their states, tasks, goals and plans in order to cooperatively succeed in missions. Communication is not always guaranteed or free, and agents need to reason about strategies to achieve optimal success and efficiency in missions under various constraints and with possibility of failures. To choose among alternative ways to accomplish tasks, our current implementation of Dec-RAE uses the UPOM planner (Patra et al. 2020). We also describe our work-in-progress on a new planner, D-UPOM, that incorporates some enhancements for planning in multi-agent environments.

Introduction

Recent work on the integration of acting and planning (Ghallab, Nau, and Traverso 2014; 2016) has advocated a hierarchical organization of an actor's deliberation functions, with online planning continually throughout the acting process. This view has led to the development of the RAE acting algorithm (Ghallab, Nau, and Traverso 2016), and to the development of three successively better planning-and-acting systems that use RAE as their acting component: APE (Patra et al. 2019b), RAE/RAEplan (Patra et al. 2019a), and RAE/UPOM (Patra et al. 2020).

To predict an action's outcome, most AI planning systems use an abstract *descriptive model* (e.g., a PDDL action definition). To perform an action, an acting system uses an *operational model* of the action – a piece of code telling the actor what to do. In systems that do both planning and acting, a key problem is the need for consistency between what the descriptive model predicts, and what the operational model says to do. The APE, RAE/RAEplan, and RAE/UPOM systems circumvent this problem by having each system's planner use the same operational model that RAE (the actor) uses. To predict the action's outcome, the planner runs the operational model in a simulated environment. This enables APE, RAE/RAEplan, and RAE/UPOM to operate effectively in environments that are partially observable, nondeterministic, and dynamically changing due to exogenous events.

A key limitation of the above work is that it is *single-agent* planning and acting. Although several of the above papers use test domains involving multiple robots, in each case the planning and acting is done by a single centralized system. In the current paper we describe our work on extending the approach to accommodate multiple agents that do their planning and acting in a decentralized fashion. Our contributions are as follows:

- We introduce Dec-RAE-UPOM, a decentralized multi-agent acting-and-planning engine that uses operational models like the ones used in RAE. It consists of two components, Dec-RAE and UPOM:
  - Dec-RAE, the acting component, is a generalization of RAE. Multiple agents can run Dec-RAE concurrently in a decentralized fashion, and can use it to perform actions and to communicate with each other.
  - UPOM, the planning component, is the same planner used in the RAE/UPOM system. UPOM uses a Monte-Carlo rollout technique based on the well-known UCT algorithm (Kocsis and Szepesvári 2006).
- We present experimental evaluations of Dec-RAE-UPOM in robot foraging problems. The results show that additional Monte-Carlo rollouts in the planning component improve the performance of the acting component in both single-agent and multi-agent settings. We also observe that communication enables coordination between agents thereby improves the performance of multi-agent foraging to a large extent.
- We describe our ongoing work on a decentralized version of UPOM, D-UPOM. UPOM in Dec-RAE does not support inter-agent plan coordination. But in D-UPOM, if an agent i needs to outsource a task τ to some other agent, i can ask

the other agents to predict how well they can accomplish τ, and outsource τ to the agent that can do the best job.

The rest of this paper is organized into sections on each of the following topics: (1) related work, (2) our formalism, (3) Dec-RAE-UPOM, (4) our experimental results, (5) our ongoing work on D-UPOM, and (6) limitations and opportunities for future work.

Related work

Hierarchically organized deliberation techniques such as HTN planning (Nau et al. 1999) and refinement acting (Ghallab, Nau, and Traverso 2016) are well-established in the AI planning literature. They have substantial advantages in working out interactions in more abstract plan spaces, thereby pruning away large portions of the more detailed search spaces (Durfee 2001).

Some AI planning researchers have been advocating a change in focus to a combination of planning and acting that incorporates 1) hierarchically organized deliberation, in which each action in a plan may be a task that may need further refinement and planning; and 2) continual planning and deliberation, in which the actor monitors, refines, extends, updates, changes and repairs its plans throughout the acting process, using both descriptive and operational models of actions (Ghallab, Nau, and Traverso 2016).

To predict an action's outcome, most AI planning systems use an abstract *descriptive model* (e.g., a precondition-and-effects model). To perform an action, an acting system must use a more-detailed *operational model* that tells what to do. In the RAE acting system (Ghallab, Nau, and Traverso 2016), these operational models are collections of *refinement methods*. A refinement method for a task t specifies *how to* perform t, i.e., it gives a procedure for accomplishing t by performing subtasks, commands and state variable assignments. The procedure may include any of the usual programming constructs: if-then-else, loops, etc. It recursively refines abstract activities into less abstract activities, ultimately producing commands to the execution platform. When several method instances are available for a task, RAE is capable of trying alternative methods in nondeterministic choices or making the choice using some heuristics.

Monte Carlo tree search (MCTS) is a promising approach for online planning because it efficiently searches over long planning horizons and is anytime (Browne et al. 2012). In treating the choice of child node to expand in the MCT as a multiarmed bandit problem, the UCT algorithm balances the tradeoff between exploration and exploitation, in order to find a near-optimal plan. The UCT-like UPOM planner (Patra et al. 2020) performs MCTS over a space of refinement trees generated using RAE's refinement methods, in order to find a near-optimal method for RAE to use for refining one of its tasks. RAE and UPOM thus constitute an integrated refinement acting-and-planning system in which both acting and planning use RAE's operational models.

Distributed problem solving is applied to a subfield of distributed artificial intelligence or multiagent systems that emphasizes on getting agents to work together well to solve problems that require collective effort (Durfee 2001). In Multi-Agent Planning, the planning process is either centralised (e.g., a master agent produces distributed plans for multiple slave agents to act upon), or decentralized (i.e., the planning process involves multiple agents) (Cardoso and Bordini 2017). Hierarchical deliberation has substantial advantages in working out interactions in more abstract plan spaces, thereby pruning away large portions of the more detailed search spaces (Durfee 2001).

The main steps of distributed hierarchical planning have been summarized in various work (Weerdt and Clement 2009; Cardoso and Bordini 2017; Durfee 2001): 1) global task (goal) refinement, decomposition of the global task into subtasks; 2) task allocation, use of task-sharing protocols to allocate tasks (goals); 3) coordination during planning, cooperative planning mechanisms that generates a globally optimal solution for the problem; 4) coordination during plan execution, mechanisms that carry out the solution, prevent conflicts, repair the plan and replan.

Dix et al. (2003) describe a formalism to integrate the HTN planning system SHOP (Nau et al. 1999) with the IMPACT multi-agent environment. While the formalism is a multi-agent system, the planning is carried out in a centralized fashion by a single agent, A-SHOP. HTN planning has been used for coordination in robot soccer (Obst and Boedecker 2006), where low-level primitive tasks are performed differently by agents depending on their roles in the team task, high-level tasks are expanded to subtasks in a centralized planner. Clement, Durfee, and Barrett (2007) developed an algorithm for hierarchical refinement planning and centralized plan co-ordination for actions with temporal extent. Summary information is derived from each high-level task in a plan hierarchy about all of its potential needs and effects under all of its potential refinements, then exchanged among agents. Coordination at abstract levels allows each agent to retain some local flexibility to refine its high-level task without jeopardizing coordination or triggering new rounds of renegotiation.

Among studies in decentralized hierarchical planning, market-based task-allocation has been applied to complex tasks that can be hierarchically decomposed (Zlot and Stentz 2006). A multi-agent model for plan synthesis that produces a global shared plan is based on unified HTN and POP approaches (Pellier and Fiorino 2007), where agents exchange proposals and counter-proposal to refine flaws. Planner9 (Magnenat, Voelkle, and Mondada 2009) is a HTN planner that considers different robots as computer clusters and distributes the planning of any task to any robot, thus takes advantage of all the available computational power using simple synchronization.

Our approach shares some similarities with reinforcement learning (RL) (Kaelbling, Littman, and Moore 1996; Sutton and Barto 1998; Geffner and Bonet 2013; Leonetti, Iocchi, and Stone 2016; Garnelo, Arulkumaran, and Shanahan 2016), since MCTS is also a typical technique in RL to increase sample efficiency. In Model-based RL, the model (e.g., system dynamics) is learned from real experience and gives rise to simulated experience. While in our work, the model which we use to construct the simulator, is given.

Decentralized partially-observable Markov decision pro-

cess (Dec-POMDP) is a framework for a team of collaborative agents to maximize a global reward based on local information. Each agent's individual policy maps from its action and observation histories to actions (Oliehoek 2012). Unfortunately, optimally solving Dec-POMDPs is NEXP-complete (Bernstein, Zilberstein, and Immerman 2013). In single-agent (i.e., MDP) domains, the options framework (Sutton, Precup, and Singh 1999) uses higher-level, temporally extended macro-actions (or options) to represent and solve problems, resulting in significant improvement in the performance. Amato et al. extend the framework to the multi-agent case by introducing a Dec-POMDP formulation with macro-actions modeled as options. It is an offline planner that generates a policy to select the best option on each state, while our approach is an planning and acting engine that selects the best refinement method for each task online.

To our knowledge, there is no prior work on decentralized refinement acting and (hierarchical) planning that uses operational models.

Formalism

Here we formalize a decentralized multi-agent refinement planning and acting domain with operational models as a tuple $\Sigma = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \{\Omega_i\}, \{\mathcal{O}_i\}, \{\mathcal{B}_i\}, \{\mathcal{C}_i\}, \{\mathcal{P}_i\}, \{\mathcal{D}_i\}, \{\mathcal{R}_i\}, \{\mathcal{M}_i\})$, where

- $\mathcal{S}$ is a set of world states the agents may be in, where we represent each state s using a state variable formulation similar to the one in Ghallab, Nau, and Traverso (2016).
- $\mathcal{I}$ is a finite set of agents,
- $\mathcal{T}$ is a finite set of tasks and events the agents may have to deal with, where each task $\tau \in \mathcal{T}$ has 0 or more than 0 relevant methods in $\{\mathcal{M}_i\}$,
- Ω_i is a finite set of observations for each agent i,
- $\mathcal{O}_i$ is a set of observation probability functions for each agent, i, $\Omega \times \mathcal{C}_i \times \mathcal{S} \rightarrow [0, 1]$,
- $\mathcal{B}_i$ is a set of belief states that agent $i \in \mathcal{I}$ may have,
- $\mathcal{C}_i$ is a finite set of primitive actions (commands) that can be carried out by the actuators and sensors on agent i's execution platform,
- $\mathcal{P}_i$ is a set of state transition probabilities, $\mathcal{S} \times \mathcal{C}_i \times \mathcal{S} \rightarrow [0, 1]$,
- $\mathcal{D}_i$ is a set of time durations, $\mathcal{S} \times \mathcal{C}_i \times \mathcal{S} \rightarrow \mathbb{R}$,
- $\mathcal{R}_i$ is a finite set of reward (or cost) functions that are associated with entering some states, i.e., $\mathcal{S} \rightarrow \mathbb{R}$,
- $\mathcal{M}_i$ is the set of refinement methods, each of which specifies how agent i would perform a task or respond to a event $\tau \in \mathcal{T}$.

A refinement method is composed of 4 elements, where 1) $head$ specifies the name and parameters of the method, where the number of parameters could be arbitrary greater than that in the task which it is related to, 3) $agent$ is the agent subject that owns (or is responsible for) the method, 3) $tasks$ indicates the task that the method is capable of refining, 3) $body$ gives a procedure to accomplish a the task by performing subtasks, commands and state variable assignments, where the procedure may include any programming constructs (e.g., if-then-else, loops, etc.). We get a refinement method instance by assigning values to the free parameters of a method.

A refinement tree (Patra et al. 2020) is composed of 3 types of nodes: 1) a *disjunction* node is a task followed by its applicable method instances; 2) a *sequence* node is a method instance m followed by all the steps; and 3) a *sampling* node for an action a has the possible nondeterministic outcomes of a as its children.

Refinement planning under this formalism is essentially a tree search procedure over the space of refinement trees in order to find a near-optimal method to use for refining a task under the context at hand.

Let us illustrate this formalism by an example, where several robots (drones and roombas) forage for some target objects (e.g., dirt) in a initially known terrain. This domain includes but is not limited to

- a set of states S that gives the positions of agents and dirt
- a set of agents $I = \{d_1, r_1, r_2\}$, where d_1 is a drone, r_1 and r_2 are roombas,
- a set of refinement methods $\mathcal{M}_{r_1} \supseteq \{$m1-cleanSet$(s)$, m2-cleanSet$(s)$, m1-clean$(s)$, m1-broadcastGoal$(g)\}$ for agent r_1,
- a set of refinement methods $\mathcal{M}_{r_2} \supseteq \{$m3-cleanSet$(s, l)$, m2-clean$(s)\}$ for agent r_2,
- a set of refinement methods $\mathcal{M}_{d_1} \supseteq \{$m1-search$(a)$, m1-planTrajectory$(a)$, m1-flyTo$(l)\}$ for agent d_1,
- a set of commands $\mathcal{C}_{d_1} \supseteq \{$observe$(l)\}$ for agent d_1,
- a set of tasks $\mathcal{T} \supseteq \{$search(a), cleanSet(s), clean(l), planTrajectory(a), flyTo$(l)\}$.

m1-search(a)
  agent: d_1
  task: search(a)
  body: trajectory $\leftarrow$ do task planTrajectory(a)
    for l in trajectory:
      do task flyTo(l)
      execute command observe(l)
      if l has dirt:
        outsource task clean(l) to agent $i \in \{r_1, r_2\}$

m1-search(a) is a method for the drone d_1 to search area a along a trajectory, perform the command observe(l) to check if an intermediate location l is dirty before outsourcing a task to a roomba i to clean up the location. Suppose roomba r_1 has one method for the task clean(l), roomba r_1 has two method for the task clean(l), there would be 3 applicable method instances in total to the task that is being outsourced, as either r_1 or r_2 could be assigned to i.

```
m1-cleanSet(s)
  agent: r_1
   task: cleanSet(s)
  body: if s is ∅:
            return
        l ← closest l ∈ s
        do task broadcastGoal(l)
        do task clean(l)
        s ← s \ l
        do task cleanSet(s)
```

```
m2-cleanSet(s)
  agent: r_1
   task: cleanSet(s)
  body: if s is ∅:
            return
        l ← random l ∈ s
        do task clean(l)
        s ← s \ l
        do task cleanSet(s)
```

Task cleanSet(s) requires a set of locations s to be cleaned. Agent r_1 has two methods that can refine the task, m1-cleanSet(s) and m2-cleanSet(s). m1-cleanSet(s) is a $greedy$ method for roomba r_1 to clean the location set s, where roomba r_1 cleans the closest location in s recursively. m2-cleanSet(s) makes random choice of the first location to clean and cleans the rest of the set recursively.

```
m3-cleanSet(s, l):
  agent: r_2
   task: cleanSet(s)
  body: if s is ∅:
            return
        do task clean(l)
        s ← s \ l
        do task cleanSet(s)
```

Agent r_2 also has a $simple$ method m3-cleanSet(s, l) that refine task cleanSet(s), where l indicates the first location to clean. l's value is automatically assigned with some predefined rules (e.g, $l \in s$). In that case, the number of applicable method instances to task cleanSet(s) for agent r_2 is $|s|$.

Dec-RAE-UPOM

RAE (Ghallab, Nau, and Traverso 2016) is a refinement acting engine that uses a collection of hierarchical refinement methods with operational model to generate and traverse a refinement tree. UPOM (Patra et al. 2020) is a UCT-like Monte-Carlo tree search simulation procedure over the space of refinement trees in order to find a near-optimal method in RAE to accomplish the task under the context at hand. A decentralized version of RAE using UPOM, Dec-RAE-UPOM is a decentralized multi-agent planning and acting engine with UCT-like planning procedure using operational models that enables heterogeneous robots to cooperatively operate in a partially-observable, non-deterministic and dynamic environment with exogenous events and concurrent tasks. As is shown in Figure 1, each agent has its own RAE, UPOM, domain knowledge, execution platform, and internal state information.

Belief, desire and intention (Rao and Georgeff 2000) represents the information, motivational, and deliberative states of the agent. Respectively, we specify 4 types of communication messages that one agent can send to another: 1) $state$, the state information obtained from the agent's sensors, i.e., its belief state set about the world which is stored in a predefined data structure; 2) $goal$, a desire that has been adopted for active pursuit by the agent; 3) $task$, a desire that the agent needs other agents to accomplish (e.g. a subtask τ in agent A's method that needs to be accomplished by agent B); 4) $plan$, the refinement tree or the estimated reward λ of its root node that is associated with the process of refining a task ($plan$ communication is not yet supported and UPOM does not require communication).

Agent are built with both actuators and sensors to send and receive communication signals. Commands are given to agents to sense the communication network, send messages, or read messages. Received messages are buffered in memory waiting for the agent to read. We acknowledge the fact that communication is neither free, nor guaranteed to succeed. Therefore, each communication command is associated with a cost and a probability of success just like any other commands.

Dec-RAE agents, without using any planner, are able to reactively coordinate their actions through $state$, $goal$ and $task$ communication. With UPOM, each agent i can plan for the selection of method instances from $\mathcal{M}_i$ to resolve a tasks locally.

Experimental Evaluation

Multi-agent Foraging is one of the canonical testbeds for cooperative multi-agent systems, in which a collection of robots has to search and transport objects to specific locations (Zedadra et al. 2017). We developed a Vacuum World Simulator (Figure 2) where multiple $roomba$ agents cooperatively clean up a finite amount of $dirt$ objects scattered randomly within an $n \times n$ grid. Each $dirt$ object is associated with a value. Each $roomba$ agent has limited amount of time budget to carry out durative actions including moving forward, turning left, turning right, picking up a dirt right beneath it, and communicating with other agents. As a preliminary experiment, actions and observations are deterministic, communication between agents is free and guaranteed. The objective is for the $roomba$ team to maximize the score (i.e., the total value from collecting $dirt$ objects) within the limit of its time budget.

We tested and compared the performance of $roomba$ agents with several different types of decision strategies (Figure 3 & 4) denoted by following remarks: 1) $Greedy$, which means the agent has a $greedy$ method, i.e., m1-cleanSet(s), that repeatedly pursues the nearest target; 2) $Simple$, which means the agent has a $simple$ method, i.e., m1-cleanSet(s, l), to pursue a list of targets; 3) $UPOM$, which indicates that the agent has the same methods as a $simple$ agent, but uses UPOM to plan for the choice of the method instances; 4) n, the number of UCT rollouts that is configured in a $UPOM$ agent; 4) $Comm$, which indicates that goal communication is enabled by doing task broadcastGoal(g), where the agent broadcasts information

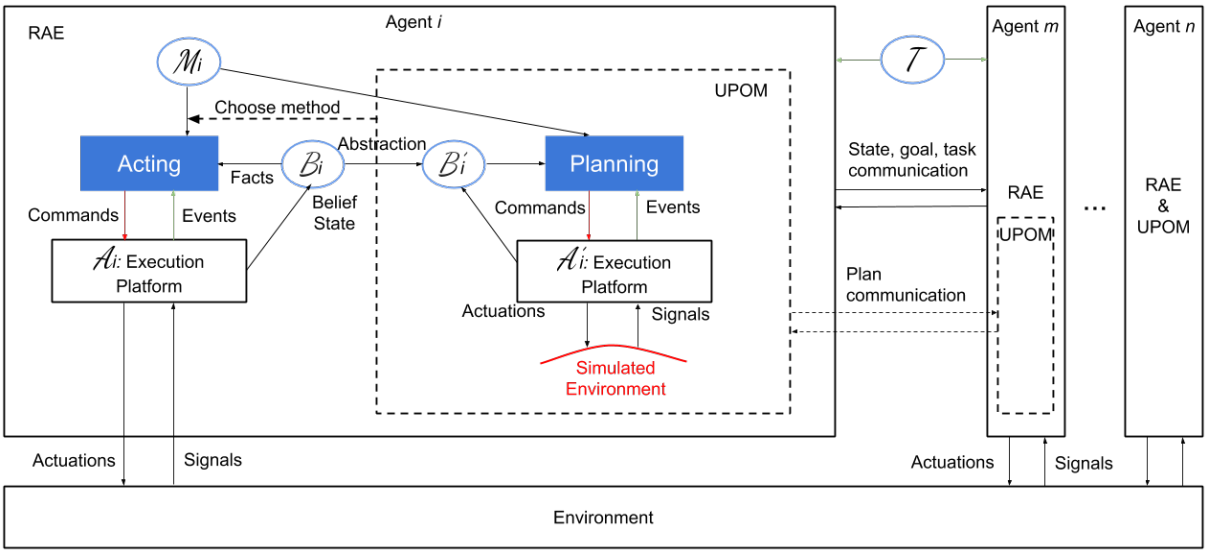


Figure 1: The system architecture of Dec-RAE-UPOM.

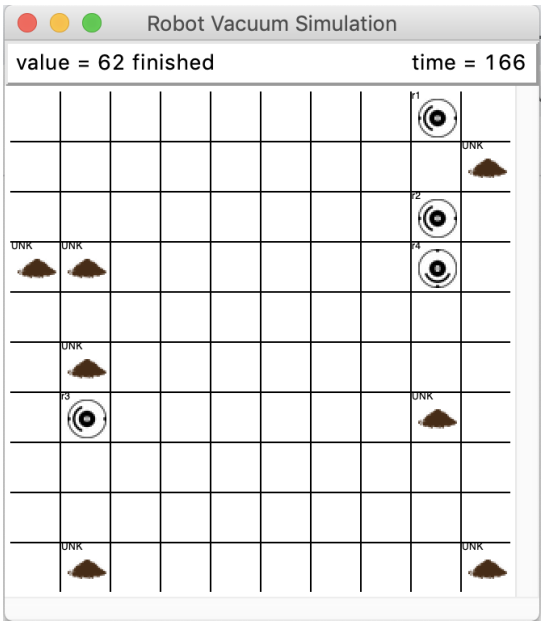


Figure 2: The Vacuum World Simulator.

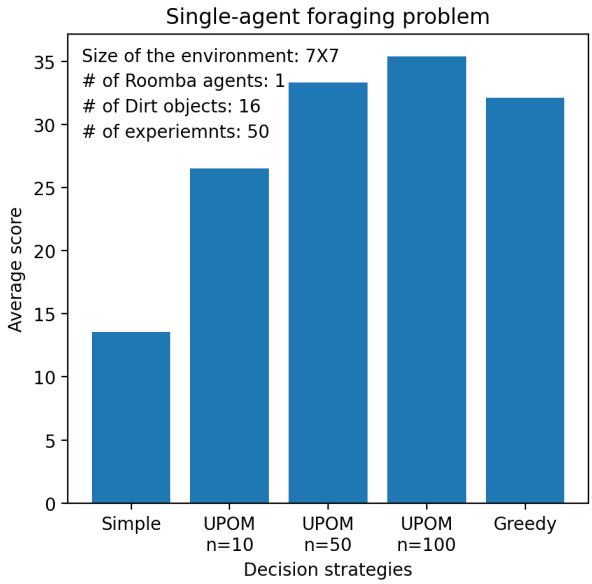


Figure 3: For each experiment, we deploy 1 $roomba$ agent along with 16 Dirt objects to a 7 $\times$ 7 grid. Each $roomba$ agent type's average score is obtained from solving 50 randomly generated problems.

about the target that it is actively pursuing. Within each experiment, all agents (if there are more than one agent) use the same strategy.

The single-agent experiments (Figure 3) show that a $UPOM$ agent performs much better than a $simple$ agent, since a reactive $simple$ agent would clean the set of locations in an arbitrary sequence, while the $UPOM$ agent tries to plan for the optimal sequence. The performance of a $UPOM$ agent further improves as the number of UCT rollouts increases, which surpasses a $greedy$ agent's performance with 50 rollouts. In multi-agent experiments (Figure 4), with goal communication enabled, agents would be aware of each other's goals, thus, are able to adjust their own goals accordingly to avoid duplication of efforts. We observe a 48.5% improvement in the $greedy$ agent team's performance with goal communication, compared to the performance without any communication. The $UPOM$ agent team with $comm$ performs slightly better than a $greedy$ agent team with $comm$, which is consistent to the result from the single-agent experiments.

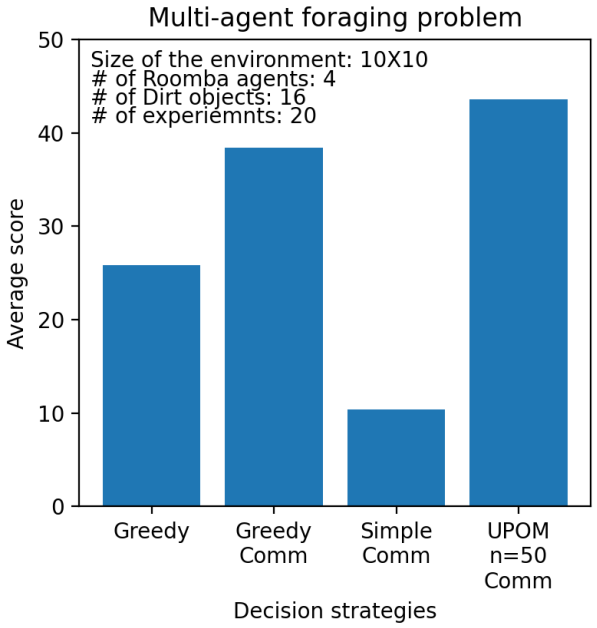


Figure 4: For each experiment, we deploy 4 *roomba* agents along with 16 Dirt objects to a 10×10 grid. Each *roomba* team's average score is obtained from solving 20 randomly generated problems.

D-UPOM

Each of the agents with Dec-RAE-UPOM has independent domain knowledge, execution platform, and state information. if an agent i needs to outsource a task τ to some other agent, without $plan$ communication, agent i has no idea how well they can accomplish τ. Therefore, agent i is only able to outsource the task to an agent that is selected randomly or based on agent i's subjective heuristics. In order to generate more optimal plans, agents need to coordinate with each other in the planning process and communicate their local plans with each other. Here we describe our work-in-progress on a decentralized version of UPOM, D-UPOM (Algorithm 1) that addresses the above issue. In D-UPOM, if an agent i needs to outsource a task τ to some other agent, i can ask other agents to predict how well they can accomplish τ, and outsource τ to the agent that can do the best job.

Firstly, we let $\gamma(s, c)$ be the set of states that may be reached after performing command c in state s. Applicable(b, τ) is the set of method instances applicable to τ in belief state b. The current context for an incoming external task τ is represented via a *refinement stack* σ which keeps track of how much further RAE has progressed in refining τ. next(σ, s) is the refinement stack resulting by performing $m[i]$ in state s, where $(\tau, m, i) = top(\sigma)$. Abstraction$(b)$ is the abstracted belief state that is used in D-UPOM's simulated environment. Responsible(m) is the agent subject that is responsible for executing method m. R(s) is the reward obtained from entering state s.

When an agent has to perform the task τ in a belief state b and a stack σ, it calls Select-Method $(b, \tau, \sigma, d_{max}, n_{ro})$

Select-Method$(b, \tau, \sigma, d_{max}, n_{ro})$:
1 $\tilde{m} \leftarrow \text{argmax}_{m \in \text{Applicable}(b,\tau)} h(\tau, m, b)$
$d \leftarrow 0$
2 **repeat**
  $d \leftarrow d + 1$
3 $b' \leftarrow$ Abstraction(b)
  for n_{ro} *times* **do**
    D-UPOM $(b', \text{push}((\tau, nil, nil), \sigma), d)$
  $\tilde{m} \leftarrow \text{argmax}_{m \in M} Q_{s,\sigma}(m)$
until $d = d_{max}$ *or* searching time is over
return $\tilde{m}$

D-UPOM(s, σ, d):
if $\sigma = \langle\rangle$ **then** return 0
$(\tau, m, i) \leftarrow \text{top}(\sigma)$
4 **if** $d = 0$ **then** return $h(\tau, m, s)$
if $m = nil$ *or* $m[i]$ *is a task* τ' **then**
  if $m = nil$ **then** $\tau' \leftarrow \tau$ # for the first task
  if $N_{s,\sigma}(\tau')$ *is not initialized yet* **then**
5 $M' \leftarrow \text{Applicable}(s, \tau')$
    if $M' = 0$ **then** return 0
    $N_{s,\sigma}(\tau') \leftarrow 0$
    for $m' \in M'$ **do**
      $N_{s,\sigma}(m') \leftarrow 0$
      $Q_{s,\sigma}(m') \leftarrow 0$
  $Untried_m \leftarrow \{m' \in M' | N_{s,\sigma}(m') = 0\}$
  if $Untried_m \neq \emptyset$ **then**
    $m_c \leftarrow$ random selection from $Untried_m$
  else $m_c \leftarrow \text{argmax}_{m \in M'} \phi(m, \tau')$
  $\sigma' \leftarrow \text{push}((\tau', m_c, 1), \text{next}(\sigma, s))$
  $a \leftarrow$ Responsible(m_c)
  if *a is self* **then** $\lambda \leftarrow$ D-UPOM$(s, \sigma', d-1)$
6 **else** $\lambda \leftarrow$ Request-Plan$(a, s, \sigma', d-1)$
7 $Q_{s,\sigma}(m_c) \leftarrow \frac{N_{s,\sigma}(m_c) \times Q_{s,\sigma}(m_c) + \lambda}{1 + N_{s,\sigma}(m_c)}$
  $N_{s,\sigma}(m_c) \leftarrow N_{s,\sigma}(m_c) + 1$
  return λ
if $m[i]$ *is an assignment* **then**
  $s' \leftarrow$ state s updated according to $m[i]$
  return D-UPOM$(s', \text{next}(\sigma, s'), d)$
if $m[i]$ *is a command* c **then**
  $s' \leftarrow \text{Sample}(s, c)$
  return $R(s') +$ D-UPOM$(s', \text{next}(\sigma, s'), d-1)$

Algorithm 1: D-UPOM and Select-Method.

(Algorithm 1) with two control parameters: n_{ro}, the number of rollouts, and d_{max}, the maximum rollout length (total number of sub-tasks and actions in a rollout). Select-Method performs an anytime progressive deepening loop calling D-UPOM n_{ro} times in a simulated environment based on the belief state b, until the rollout length reaches d_{max} or the search is interrupted. The selected method instance $\tilde{m}$ is initialized according to a heuristic h (line 1).

Just like UPOM, D-UPOM performs one UCT rollout recursively down the refinement tree until depth d is reached for stack σ. During the recursion, if another agent a is re-

sponsible for executing method m_c, one needs to request the agent to plan for m_c by calling Request-Plan$(a, s, \sigma', d-1)$ (line 6). Agent a is supposed to receive the request, use its own D-UPOM to perform one UCT rollout further down the refinement tree recursively, and send back the resulting estimated total reward λ from executing m_c. Request-Plan will return 0 if it times out.

D-UPOM naturally supports market-based task allocation: when the task τ' is potentially outsourced to different agents who are capable of accomplishing it, each agent plans for τ' using its methods and return the corresponding estimated rewards, the agent who has a method that obtains the highest reward will be chosen to accomplish τ'.

As a work-in-progress, D-UPOM has not yet been programmed or tested. We will further develop its theory, program and experiments in our future work.

Discussion

In this paper we have described Dec-RAE-UPOM, a system for decentralized multi-agent refinement planning and acting that uses operational models similar to the ones used in the RAE and RAE/UPOM systems. In our evaluations of Dec-RAE-UPOM's performance in robot foraging problems using the Vacuum World Simulator, the results show that the system's performance is improved by performing additional Monte-Carlo rollouts in UPOM, and that communication between agents also improves the performance. We have also described our work-in-progress on the D-UPOM planner, a decentralized version of UPOM that is expected to generate more optimal plans by exploiting $plan$ information obtained from other agents.

Multi-agent systems are usually affected by the combinatorial explosion of the state space due to increased amount of agents. A decentralized planner avoids this problem by making each agent plan locally and communicate their plans with each other in order to cooperate. However, functionalities such as action synchronization and conflict resolution between decentralized agents are not currently supported by our system.

The $plan$ communication in D-UPOM only delivers minimum information, i.e., the total reward obtained from sampling the method. In a similar planner, D-UCB in decentralized Monte Carlo Tree search (Dec-MCTS) (Best et al. 2018), agents exchange more explicit plan information back and forth so the planner in each agent is able to sample all the other agents' actions according to their plans. D-UPOM has the same potential in exchanging more plan information such as a explicit refinement tree and its associated Q values, which might help with plan merging, conflict resolution and action synchronization. Our formalism represents action durations, and we intend to reason about plan merging, conflict resolution and action synchronization as a temporal planning and scheduling problem in our future work.

In our experiments, communication commands have 0 cost (reward) and are guaranteed to succeed. We have not done enough investigations in cases where communication is not always guaranteed or free, and agents might need to proactively search for communication signals (e.g., by going to a high ground where there is higher chance to re-establish communication with others). A broader question that automotive agents needs to decide is who, when, how and what to communicate (Balch and Arkin 1970; Wei, Hindriks, and Jonker 2014). As an example, Dec-MCTS reasons about the value of communication messages to decide when and to whom each robot should communicate in order to minimize the communication cost. We hope our work could be developed to be more resilient and intelligent in terms of communication.

Since UPOM has been integrated with neural networks that learn to choose methods and approximate heuristics, we are also interested in extending learning to our future work.

Acknowledgements

This work has been supported in part by DARPA task order HR001119F0057, Lockheed Martin research agreement MRA17001006, NRL grant N00173191G001, and ONR grant N000142012257. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Amato, C.; Konidaris, G.; Kaelbling, L.; and How, J. 2019. Modeling and planning with macro-actions in decentralized pomdps. *Journal of Artificial Intelligence Research* 64:817–859.

Balch, T., and Arkin, R. 1970. Communication in reactive multiagent robotic systems. *Autonomous Robots* 1.

Bernstein, D. S.; Zilberstein, S.; and Immerman, N. 2013. The complexity of decentralized control of markov decision processes.

Best, G.; Forrai, M.; Mettu, R. R.; and Fitch, R. 2018. Planning-aware communication for decentralised multi-robot coordination. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 1050–1057.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Cardoso, R., and Bordini, R. 2017. A multi-agent extension of a hierarchical task network planning formalism. *ADCAIJ* 6:5.

Clement, B.; Durfee, E.; and Barrett, A. 2007. Abstract reasoning for planning and coordination. *J. Artificial Intelligence Research (JAIR)* 28:453–515.

Dix, J.; Muñoz-Avila, H.; Nau, D. S.; and Zhang, L. 2003. Impacting shop: Putting an ai planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence* 37(4):381–407.

Durfee, E. H. 2001. Distributed problem solving and planning. In Luck, M.; Mařík, V.; Štěpánková, O.; and Trappl, R., eds., *Multi-Agent Systems and Applications: European Agent Systems Summer School, EASSS 2001*, 118–149. Springer.

Garnelo, M.; Arulkumaran, K.; and Shanahan, M. 2016. Towards deep symbolic reinforcement learning. *CoRR* abs/1609.05518.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actors view of automated planning and acting: A position paper. *Artificial Intelligence* 208:1 – 17.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *JAIR* 4:237–285.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. 282–293.

Leonetti, M.; Iocchi, L.; and Stone, P. 2016. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. 241:103–130.

Magnenat, S.; Voelkle, M.; and Mondada, F. 2009. Planner9, a HTN planner distributed on groups of miniature mobile robots.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, 968–973. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Obst, O., and Boedecker, J. 2006. Flexible coordination of multiagent team behavior using HTN planning. In Bredenfeld, A.; Jacoff, A.; Noda, I.; and Takahashi, Y., eds., *RoboCup 2005: Robot Soccer World Cup IX*, 521–528. Berlin, Heidelberg: Springer.

Oliehoek, F. A. 2012. *Decentralized POMDPs*. Berlin, Heidelberg: Springer Berlin Heidelberg. 471–503.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019a. Acting and planning using operational models. *Proceedings of the AAAI Conference on Artificial Intelligence* 33:7691–7698.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019b. APE: An acting and planning engine. *Advances in Cognitive Systems* 7.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating acting, planning and learning in hierarchical operational models.

Pellier, D., and Fiorino, H. 2007. A unified framework based on HTN and POP approaches for multi-agent planning. In *2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07)*, 285–288.

Rao, A., and Georgeff, M. 2000. Bdi agents: From theory to practice.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1):181 – 211.

Weerdt, M., and Clement, B. 2009. Introduction to planning in multiagent systems. *Multiagent and Grid Systems* 5:345–355.

Wei, C.; Hindriks, K.; and Jonker, C. 2014. The role of communication in coordination protocols for cooperative robot teams. In *ICAART 2014 - Proceedings of the 6th International Conference on Agents and Artificial Intelligence*, volume 2.

Zedadra, O.; Jouandeau, N.; Seridi, H.; and Fortino, G. 2017. Multi-agent foraging: state-of-the-art and research challenges. *Complex Adaptive Systems Modeling* 5:1–24.

Zlot, R., and Stentz, A. 2006. Market-based multirobot coordination for complex tasks. *The International Journal of Robotics Research* 25(1):73–101.

Planning for Cooperative Multiple Agents with Sparse Interaction Constraints

Guy Revach,[1] **Nir Greshler,**[2] **Nahum Shimkin**[2]
[1] The Institute for Signal and Information Processing (ISI),
Department of Information Technology and Electrical Engineering (D-ITET), ETH Zurich, Switzerland
[2] Viterbi Faculty of Electrical Engineering, Technion, Haifa, Israel
revach@isi.ee.ethz.ch, nirgreshler@campus.technion.ac.il, shimkin@ee.technion.ac.il

Abstract

We consider the problem of cooperative multi-agent planning (MAP) in a deterministic environment, with a completely observable state. Most tractable algorithms for MAP problems assume *sparse interactions* among agents and exploitable problem structure. We consider a specific model for representing interactions among agents using *soft cooperation constraints (SCC)*, which enables a compact representation of symmetric dependencies. We present a two-step planning algorithm that breaks down a multi-agent problem with K agents, to multiple instances of independent single-agent problems, such that the aggregation of the single-agent plans is optimal for the group. We propose an efficient algorithm for computing the single-agent optimal plan under a given set of soft constraints, denoted as the *response function*. We then utilize a well-known graphical model for efficient min-sum optimization in order to find the optimal aggregation of the single agent response functions. The proposed planning algorithm is complete, optimal, and effective when interactions among the agents are sparse. We further indicate some useful extensions to the basic SCC formulation presented here.

1 Introduction

The problem of cooperative multi-agent planning (MAP) is motivated by many real-world applications in a variety of domains, such as military, logistics, and search-and-rescue. In these problems, agents must coordinate their decisions to maximize their (joint) team value. When the state of the environment and all agents is fully-observable by each agent, the planning problem can be formalized as a multi-agent Markov decision process (MMDP, Boutilier 1996). However, these models suffer from exponential increase in the size of the state and action spaces in the number of agents, which makes them computationally intractable in general. Specific structural assumptions are therefore required for an optimal solution to be feasible.

An important class of problems concerns high-level planning problems, where agents are essentially independent except for a prescribed set of possible interactions that can facilitate the plan execution. These types of problems are typically characterized by *loose coupling* and *sparse interactions* between agents, and some models exploit this fact to develop efficient algorithms. The complexity of such algorithms is often described by means of the problem coupling level. For instance, (Nissim, Brafman, and Domshlak 2010) propose a fully distributed planning algorithm, based on the MA-STRIPS (Brafman and Domshlak 2008) model, and (Melo and Veloso 2011) propose approximate algorithms based on the decentralized sparse-interaction MDPs model.

Another common approach is to exploit the problem structure by using a compact representation with factored models. An example of such a representation is the *coordination graph* (Guestrin, Koller, and Parr 2002), also referred to as *interaction graph* (Nair et al. 2005) or *collaborative graphical games* (Oliehoek, Whiteson, and Spaan 2012), which is solved using a graph-based optimization method, such as variable elimination (VE) (Guestrin, Koller, and Parr 2002; Larrosa and Dechter 2003), or by distributed methods as investigated in the field of distributed constraint optimization problem (DCOPs, Fioretto, Pontelli, and Yeoh 2018).

It is also possible to exploit locality of interactions (Oliehoek et al. 2008; Melo and Veloso 2011) and reward structure in transition-independent models, both centralized (Scharpff et al. 2016) and decentralized (Becker et al. 2004). More specifically, in (Scharpff et al. 2016) reward dependencies are represented using conditional return graphs (CRGs) which are solved by a branch-and-bound policy search algorithm. In (Becker et al. 2004) a general formulation is suggested to represent the reward structure, using the notion of *events*. A coverage set algorithm is presented to find optimal policies. Scalability can often be improved even on more complex models, such as Network Distributed Partially Observable MDP (ND-POMDP), by leveraging sparse and structured interactions among agents. For example, the CBDP (Kumar and Zilberstein 2009) algorithm is exponential only in the width of agents interaction graph.

In this paper, we focus on the multi-agent planning problem in a *deterministic* environment, where interactions between agents are symmetric and sparse. Possible interactions are captured using a notion of *soft cooperation constraints (SCC)*, where agents can affect the cost function by jointly satisfying prescribed constraints in state and time.

This formulation is akin to the event-based formulation of (Becker et al. 2004), although less general to allow more specific and explicit computation schemes for each agent.

Based on the SCC model, we present a complete and optimal two-step planning algorithm, effective mostly in cases where interactions among agents are sparse. It is a dynamic programming (DP)-based algorithm, that decouples a multi-agent problem with K agents to K independent single-agent problems, such that the aggregation of the single-agent plans is optimal for the group. More specifically, in the first step we independently compute each agent's *response function*, which is its optimal plan with respect to all possible assignments of the timing variables of its associated constraints. We present an explicit algorithm for computing the response function, and provide a detailed complexity analysis. The second step is a centralized global *plan merging*, in which an optimal assignment to the timing variables is found under the *minimum-sum* objective. A *factor graph*, which captures dependencies among cooperative agents and exploits the internal structure of the problem, is applied to the problem with a variable elimination algorithm for efficient min-sum optimization.

Complexity analysis shows that the proposed algorithm is linear in the number of agents, polynomial in the span of the time horizon, and depends exponentially only on the number of interactions among agents.

We present a simulation implementing our proposed algorithm on a specific multi-agent planning problem. Our simulations show that the algorithm is efficient for this particular multi-agent setup and scales well in the number of agents compared to a standard solution.

We finally outline possible extensions to our model, to represent more complex cooperation constraints. For details of these extensions we refer (Revach 2018).

The remainder of the paper is organized as follows. In section 2 we present the model used and the formulation of SCC. Section 3 presents a detailed description and implementation of our algorithm, followed by a complexity analysis. In section 4 we present experimental results for our algorithm. In section 5 we present an extension to our model to include asymmetric interactions between agents. Section 6 concludes the paper and suggests directions for extensions and future research.

2 Model

We consider the finite horizon multi-agent deterministic planning problem. Our starting point is an MMDP with a factored state space, defined by a tuple $\langle \mathcal{T}, \mathbf{G}, \mathcal{S}, \mathcal{A}, \mathcal{H}, \mathcal{C}, \sigma_I, \sigma_* \rangle$, where

- $\mathcal{T} = \{0, ..., T\}$ is the time domain of length T.
- $\mathbf{G} = \{\mathbf{g}_1, \mathbf{g}_2, ..., \mathbf{g}_K\}$ is a set of K agents.
- $\mathcal{S} = \mathcal{S}_1 \times ... \times \mathcal{S}_K$ is a finite state space, factored across agents, where $\mathcal{S}_k$ is the state space of agent $\mathbf{g}_k$.
- $\mathcal{A} = \mathcal{A}_1 \times ... \times \mathcal{A}_K$ is a joint action space, similarly factored across agents.
- $\mathcal{H} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a deterministic transition function.
- $\mathcal{C} : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \cup \{\infty\}$ is a real-valued cost function.
- $\sigma_I \in \mathcal{S}$ is the initial state $\vec{s}_0$, and $\sigma_* \in \mathcal{S}$ the goal state.

Our objective is to find an optimal group policy $\vec{\pi}^*$ such that $\mathbf{J}^{\vec{\pi}}$ is minimal, i.e., $\vec{\pi}^* \in \arg\min_{\vec{\pi} \in \Pi^K} \mathbf{J}^{\vec{\pi}}$, where $\vec{\pi} = (\pi_1, ..., \pi_K)$ is the joint policy, $\mathbf{J}^{\vec{\pi}}$ is the aggregate cumulative cost defined by

$$\mathbf{J}^{\vec{\pi}} = \sum_{t=0}^{T-1} \mathcal{C}\left(\vec{s}_t, \vec{a}_t\right) \tag{1}$$

if $\vec{s}_T = \sigma_*$, and $\mathbf{J}^{\vec{\pi}} = \infty$ otherwise. Here $\vec{a}_t = (a_{t,1}, a_{t,2}, ..., a_{t,K}) \in \mathcal{A}$ is the joint action at time t, such that $a_{t,k} = \pi_k\left(s_t, t\right)$.

We next describe the sparse interactions structure. We first assume transition and cost independence across agents, namely

$$\mathcal{H}\left(\vec{s}, \vec{a}\right) = \left(\mathcal{H}_1\left(s_1, a_1\right), \ldots, \mathcal{H}_K\left(s_K, a_K\right)\right) \tag{2}$$

and

$$\mathcal{C}\left(\vec{s}, \vec{a}\right) = \sum_{k=1}^{K} \mathcal{C}_k\left(s_k, a_k\right) \tag{3}$$

Coupling between agents is introduced via a set $\Psi = \{\psi_1, ..., \psi_L\}$ of *soft cooperation constraints*. Each constraint ψ_ℓ defines a single opportunity for cooperative interaction between agents. In particular, a constraint ψ_ℓ, $\ell \in \{1, ..., L\}$, is specified by the following tuple:

$$\psi_\ell = \left\langle \mathbf{G}_\ell, \Sigma_\ell, \mathcal{C}_\ell^-, \mathcal{T}_\ell \right\rangle \tag{4}$$

where

- $\mathbf{G}_\ell = \{\mathbf{g}_k,\ k \in K_\ell\}$, with $K_\ell = (k_{\ell,1}, \ldots, k_{\ell,n(\ell)})$, is the set of $n(\ell)$ agents interacting in constraint ψ_ℓ.
- $\Sigma_\ell = \{\sigma_{\ell,k}, k \in K_\ell\}$, with $\sigma_{\ell,k} \in \mathcal{S}_k$, is a set of local interaction states. Namely, for the constraint to hold, agent k is required to be in state $\sigma_{\ell,k}$ at some prescribed time.
- $\mathcal{C}_\ell^-$ is a (reduced) immediate cost for the group for interaction, applicable when the constraint is satisfied (see equation 5).
- $\mathcal{T}_\ell$ is the constraint time domain; i.e., it is a subset of time instances at which the interaction may take place: $\mathcal{T}_\ell \subseteq \{0, 1, ..., T-1\} \cup T_\emptyset$. Here $T_\emptyset$ is a special notation for the *null assignment*, where the constraint is not satisfied, i.e., there is no interaction.

Note that the SCC formulation can be extended to represent more general constraints. For instance, a constraint can have a set of time domains, one for each agent, such that each agent interacts at a different time. Moreover, a constraint can have a subset of interaction states (instead of a single state). While the ideas are similar, for concreteness and brevity we leave these extensions to future work.

2.1 Interaction-Dependent Cost

Agents are coupled only via the constraint set Ψ. Therefore, the group cost depends on the constraints satisfied, where each satisfied constraint ψ_ℓ represents an interaction which applies the group a reduced cost $\mathcal{C}_\ell^-$. We now describe the structure of the group cost under this formulation.

Let $\tau_\ell \in \mathcal{T}_\ell$ be an *interaction timing variable* that defines the timing of the interaction under constraint ψ_ℓ. For a given assignment to the timing variable τ_ℓ, we define an indicator function that is true if all interacting agents in $\mathbf{G}_\ell$ satisfy constraint ψ_ℓ:

$$\hat{\psi}_\ell(\tau_\ell; \vec{\pi}) = \mathbb{I}_{\{\tau_\ell \neq T_\emptyset\}} \prod_{k \in K_\ell} \mathbb{I}_{\{s_{\tau_\ell,k} = \sigma_{\ell,k}\}} \quad (5)$$

where $\mathbb{I}_A$ is the $0/1$ indicator of event A. Namely, constraint ψ_ℓ is satisfied given $\tau_\ell = \tau$ if $\tau \neq T_\emptyset$ and all interacting agents in ψ_ℓ arrive at their interaction state at time τ.

Furthermore, $\vec{\tau}$ is the interaction vector, and $\mathcal{D}$ is its domain, i.e., the cross space of all constraint time domains:

$$\vec{\tau} = (\tau_1, \tau_2, ..., \tau_L) \in \mathcal{T}_1 \times \mathcal{T}_2 \times ... \times \mathcal{T}_L \triangleq \mathcal{D} \quad (6)$$

$\vec{\tau}_k$ is the timing vector of all constraints involving agent $\mathbf{g}_k$ (with domain $\mathcal{D}_k$).

Under this new formulation, given an initial state $\sigma_I \in \mathcal{S}$, a goal state $\sigma_* \in \mathcal{S}$, and a constraint set Ψ, our objective is to find the optimal group policy where $\mathbf{J}^{\vec{\pi}}$ in equation 1 is now

$$\mathbf{J}^{\vec{\pi}}(\vec{\tau}) = \sum_{t=0}^{T-1} \sum_{k=1}^{K} \mathcal{C}_{0,k}(s_{t,k}, a_{t,k}) + \sum_{\ell=1}^{L} \hat{\psi}_\ell(\tau_\ell; \vec{\pi}) \mathcal{C}_\ell^- \quad (7)$$

where $\mathcal{C}_{0,k}$ is the single-agent independent immediate cost with no consideration of interactions. Namely, it is the sum of all agents' independent immediate cost plus the sum of the reduced costs of all satisfied constraints.

Note that now the multi-agent optimal policy $\vec{\pi}$ is a parametric policy with respect to timing variables, and the aggregate cumulative cost $\mathbf{J}^{\vec{\pi}}$ is a function of the timing variables. Effectively, there may be a different optimal policy for each assignment of timing variables. Furthermore, $\mathbf{J}_k^*(\vec{\tau}_k)$ is the optimal response function (i.e., the optimal cumulative cost) for agent $\mathbf{g}_k$ given an assignment of the timing vector $\vec{\tau}$.

Our objective is to minimize the multi-agent cumulative cost under L interaction constraints:

$$\mathbf{J}^* = \min_{\vec{\tau} \in \mathcal{D}} \min_{\vec{\pi} \in \Pi^K} \left\{ \mathbf{J}^{\vec{\pi}}(\vec{\tau}) \right\} \quad (8)$$

where $\mathbf{J}^{\vec{\pi}}(\vec{\tau})$, defined by equation 7, is decomposable, and where each single agent cost function depends only on the single agent policy. Therefore, we may switch the order of summation to compute independently for each agent:

$$\mathbf{J}^{\vec{\pi}}(\vec{\tau}) = \sum_{k=1}^{K} \sum_{t=0}^{T-1} \mathcal{C}_{0,k}(s_{t,k}, a_{t,k}) + \sum_{\ell=1}^{L} \hat{\psi}_\ell(\tau_\ell; \vec{\pi}) \mathcal{C}_\ell^- \quad (9)$$

provided that $\vec{s}_T = \sigma_*$ and $\mathbf{J}^{\vec{\pi}}(\vec{\tau}) = \infty$ otherwise.

We can then minimize each single agent cost independently for any given assignment of the timing vector $\vec{\tau} \in \mathcal{D}$ (and specifically $\vec{\tau}_k$ for each agent $\mathbf{g}_k$). After the optimal single agent response functions are found, we need to find the optimal assignment for the timing variables. Let us observe that the multi-agent problem decomposition results in a min-sum optimization problem:

$$\vec{\tau}^* \in \arg\min_{\vec{\tau} \in \mathcal{D}} \sum_{k=1}^{K} \mathbf{J}_k^*(\vec{\tau}_k) \quad (10)$$

that is, the sum of optimal response functions. We can use this structure to our advantage by applying an efficient optimization algorithm.

3 *DIPLOMA* - Distributed Planning and Optimization Algorithm for Multiple Agents

In this section we present the *DIstributed PLanning and Optimization algorithm for Multiple Agents (DIPLOMA)*, which addresses the previous multi-agent interaction model and optimizes cost and policy. Using this model, we are able to decompose a global multi-agent planning problem into a two-step problem. First, K distributed independent single-agent planning problems are solved. Second, we optimize the global solution with respect to the cooperation constraints by selecting a plan for each agent.

We now describe the steps of our proposed algorithm, presented in algorithm 1:

1. **Response Function Computation**

   For every agent $\mathbf{g}_k \in \mathbf{G}$, compute the single agent response function independently,

   $$\forall \vec{\tau}_k \in \mathcal{D}_k \ , \ \mathbf{J}_k^*(\vec{\tau}_k) = \min_{\pi_k \in \Pi_k} \mathbf{J}_k^{\pi_k}(\vec{\tau}_k) \quad (11)$$

   It may be computed using various dynamic programming algorithms, and more specifically using the algorithms described next, in detail. This step can be parallelized over agents.

2. **Plan Merging**

   Compute the optimal total multi-agent cost by minimizing the sum single agent response with respect to the constraint variables. More specifically:

   $$\mathbf{J}^* = \min_{\vec{\tau} \in \mathcal{D}} \sum_{k=1}^{K} \mathbf{J}_k^*(\vec{\tau}_k) \quad (12)$$

   The minimization process can be carried out efficiently using factor graph modeling and a variable elimination algorithm, as described below. Let $\vec{\tau}^*$ denote the optimal assignment of the constraint variables.

3. **Policy Backtracking**

   (a) For every agent $\mathbf{g}_k \in \mathbf{G}$, backtrack the single agent optimal policy independently:

   $$\pi_k^* \in \arg\min_{\pi_k \in \Pi_k} \mathbf{J}_k^{\pi_k}(\vec{\tau}_k^*) \quad (13)$$

   (b) The global optimal multi-agent policy is then given by

   $$\vec{\pi}^* = \{\pi_1^*, \pi_2^*, ..., \pi_k^*, ..., \pi_K^*\} \quad (14)$$

Algorithm 1 DIPLOMA

returns $\vec{\pi}^*$, the optimal group policy
inputs: MMDP, Ψ
for all $\mathbf{g}_k \in \mathbf{G}$ **do** ▷ response function computation
  for all $\vec{\tau}_k \in \mathcal{D}_k$ **do**
    $\mathbf{J}_k^*(\vec{\tau}_k) = \min_{\pi_k \in \Pi_k} \mathbf{J}_k^{\pi_k}(\vec{\tau}_k)$
$\mathbf{J}^* = \min_{\vec{\tau} \in \mathcal{D}} \sum_{k=1}^{K} \mathbf{J}_k^*(\vec{\tau}_k)$ ▷ plan merging
$\vec{\tau}^* \in \arg\min_{\vec{\tau} \in \mathcal{D}} \sum_{k=1}^{K} \mathbf{J}_k^*(\vec{\tau}_k)$
for all $\mathbf{g}_k \in \mathbf{G}$ **do**
  $\pi_k^* \in \arg\min_{\pi_k \in \Pi_k} \mathbf{J}_k^{\pi_k}(\vec{\tau}_k^*)$
$\vec{\pi}^* = \{\pi_1^*, \pi_2^*, ..., \pi_k^*, ..., \pi_K^*\}$
return $\vec{\pi}^*$

3.1 Response Function Computation

The main step of our proposed algorithm is computing the single agent response function with respect to constraint timing variables. We now describe algorithms to compute $\mathbf{J}_k^*$ efficiently for each agent. Before describing the algorithms in detail, we present a few basic definitions and notations:

- The algorithms presented are from a single agent perspective; therefore, we omit the index k from the notation wherever possible.
- $\mathcal{V}^*(\sigma, \sigma_*, \tau)$, the *cost-to-state*, is the optimal cumulative cost from state σ at time $t = \tau$ to the target state σ_* in $T - \tau$ time steps.
- $\mathcal{J}^*(\sigma_I, \sigma, \tau)$, the *cost-from-state*, is the optimal cumulative cost from initial state σ_I at time $t = 0$ to state σ in τ time steps, and $\mathbf{J}^* = \mathcal{J}^*(\sigma_I, \sigma_*, T)$.
- More generally, $\mathcal{V}_\diamondsuit^*(s_I, s_g, \tau_I, \tau_g)$ is the optimal cumulative cost from state s_I at time $t = \tau_I$ to state s_g at time $t = \tau_g$ in $\tau_g - \tau_I$ time steps.

We start with the following computation in algorithm 2 of the cost-to-state and cost-from-state. The result is required only for intermediate states in the agent's constraint set. A natural implementation by Dynamic Programming (DP) computes these costs via a single pass for all states and times.

Algorithm 2 Costs to and from states

for all $\tau \in \{1, .., T\}$ and $\sigma \in \{\sigma_\ell\}$ **do**
  Compute $\mathcal{J}^*(\sigma_I, \sigma, \tau)$ iteratively using DP.
for all $\tau \in \{T-1, .., 0\}$ $\sigma \in \{\sigma_\ell\}$ **do**
  Compute $\mathcal{V}^*(\sigma, \sigma_*, \tau)$ iteratively using DP.
Cache all results for later use.

The single agent response function is the optimal cumulative cost with respect to the timing variables, i.e., the optimal plan from the initial state to the goal state, while satisfying the constraints in times specified by the timing variables. To simplify the presentation, we start by showing how to compute the single-agent response function with a single interaction (i.e., a single constraint), and then follow with the general case of L interactions.

Let τ_ℓ be a single timing variable (i.e., $L = \ell = 1$), and $\mathbf{J}_\sigma^*(\tau_\ell)$ the optimal cumulative cost from initial state σ_I to goal state σ_* in T time steps, via the intermediate state σ_ℓ; i.e., $s_{\tau_\ell} = \sigma_\ell$. For a given assignment of τ_ℓ, we can compute the response function in this simple case as:

$$\mathbf{J}_\sigma^*(\tau) = \begin{cases} \mathbf{J}^*, & \tau = T_\emptyset \\ \mathcal{J}^*(\sigma_I, \sigma_\ell, \tau) + \mathcal{V}^*(\sigma_\ell, \sigma_*, \tau), & \text{otherwise} \end{cases} \quad (15)$$

Namely, it is computed by two parts: planning from the initial state σ_I in time $t = 0$ to the constraint state σ_ℓ in time $t = \tau_\ell$, and from the latter to the goal state at T (i.e., for $T - \tau$ time steps). If $\tau_\ell = T_\emptyset$, the constraint is not imposed. Hence, $\mathbf{J}_\sigma^*(\tau_\ell) = \mathbf{J}^*$, i.e., the optimal cost with no consideration of interactions.

The generalization for $L = L_k$ constraints (the number of constraints agent k is involved in) follows the same idea. We need to compute the response function $\mathbf{J}_{\sigma_1,...,\sigma_L}^*(\tau_1, ..., \tau_L)$ for every assignment of L timing variables. We present an incremental scheme that efficiently avoids repeated computation of given segments:

1. Pre-compute the state-to-state cost functions by dynamic programming, and cache the results for later use:
   1.1. Apply algorithm 2.
   1.2. Pre-compute $\mathcal{V}_\diamondsuit^*$ using algorithm 3.
2. Build the response function from the bottom up using the previously cached values that were pre-computed in the previous step, using algorithms 4 and 5. For simplicity we use the following concise notation, for $1 \le \ell \le L$:

$$\mathbf{J}^*\{\ell\} \triangleq \mathbf{J}_{\sigma_1,...,\sigma_\ell}^*(\tau_1, ..., \tau_\ell) \quad (16)$$

   In algorithm 5 we show how to compute $\mathbf{J}^*\{\ell+1\}$ for all assignments to $\tau_{\ell+1}$, given $\mathbf{J}^*\{\ell\}$ for a specific assignment to $\tau_1, \ldots, \tau_\ell$. The idea is essentially to replace $\mathcal{V}_\diamondsuit^*(\sigma_{\ell_1}, \sigma_{\ell_2}, \tau_{\ell_1}, \tau_{\ell_2})$ by the sum $\mathcal{V}_\diamondsuit^*(\sigma_{\ell_1}, \sigma_{\ell+1}, \tau_{\ell_1}, \tau_{\ell+1}) + \mathcal{V}_\diamondsuit^*(\sigma_{\ell+1}, \sigma_{\ell_2}, \tau_{\ell+1}, \tau_{\ell_2})$ when constraint $\ell+1$ is added with timing assignment $\tau_{\ell+1}$ between existing τ_{ℓ_1} and τ_{ℓ_2}.

Algorithm 3 Multiple constraint response - step 1.2

for all $\sigma_i, \sigma_j \in \{\sigma_1, \sigma_2, ..., \sigma_L\}$ **do**
  for all $\tau_i \in \mathcal{T}_i$ **do**
    for all $\tau_j \in \mathcal{T}_j$, $\tau_j > \tau_i$ **do**
      Compute $\mathcal{V}_\diamondsuit^*(\sigma_i, \sigma_j, \tau_i, \tau_j)$
      Cache the results for later use.

Algorithm 4 Multiple constraint response - step 2

for all $\ell \in \{1, 2, ..., L-1\}$ **do**
  for all $\tau_1, \tau_2, ..., \tau_\ell$ **do**
    For a given assignment to $\tau_1, \tau_2, ..., \tau_\ell$
    Such that $\tau_{i_1} \le \tau_{i_2} \le ... \le \tau_{i_\ell}$
    Compute $\mathbf{J}^*\{\ell+1\}$ from $\mathbf{J}^*\{\ell\}$ for all $\tau_{\ell+1} \in \mathcal{T}_{\ell+1}$, using Algorithm 5

Algorithm 5 Multiple constraint response - inner algorithm

Assume $\tau_{i_1} \leq \tau_{i_2} \leq ... \leq \tau_{i_\ell}$
$p = 1$ ▷ Initialize pivot index
$b = 1$ ▷ Set *baseline* flag
for all $\tau_{\ell+1} \in \{0, 1, ..., T-1\}$ **do**
  if $p = 1$ **then**
    if $b = 1$ **then**
      $\mathbf{J}^*_{base} = \mathbf{J}^*\{\ell\} - \mathcal{J}^*(\sigma_I, \sigma_{i_1}, \tau_{i_1})$ ▷ Initialize a *baseline* value for $\mathbf{J}^*\{\ell+1\}$
      $b = 0$ ▷ Reset *baseline* flag
    if $\tau_{\ell+1} < \tau_{i_1}$ **then**
      $\mathbf{J}^*\{\ell+1\} = \mathbf{J}^*_{base} + \mathcal{J}^*(\sigma_I, \sigma_{\ell+1}, \tau_{\ell+1}) + \mathcal{V}^*_\Diamond(\sigma_{\ell+1}, \sigma_{i_1}, \tau_{\ell+1}, \tau_{i_1})$
    else ▷ $\tau_{\ell+1} = \tau_{i_1}$
      $\mathbf{J}^*\{\ell+1\} = \infty$ ▷ There is no valid plan that meets the constraints
      $p = 2$ ▷ Increment pivot index
      $b = 1$ ▷ Set *baseline* flag
  else if $1 < p \leq \ell$ **then**
    if $b = 1$ **then**
      $\mathbf{J}^*_{base} = \mathbf{J}^*\{\ell\} - \mathcal{V}^*_\Diamond(\sigma_{i_{p-1}}, \sigma_{i_p}, \tau_{i_{p-1}}, \tau_{i_p})$
      $b = 0$ ▷ Reset *baseline* flag
    if $\tau_{\ell+1} < \tau_{i_p}$ **then**
      $\mathbf{J}^*\{\ell+1\} = \mathbf{J}^*_{base} + \mathcal{V}^*_\Diamond(\sigma_{i_{p-1}}, \sigma_{\ell+1}, \tau_{i_{p-1}}, \tau_{\ell+1}) + \mathcal{V}^*_\Diamond(\sigma_{\ell+1}, \sigma_{i_p}, \tau_{\ell+1}, \tau_{i_p})$
    else ▷ $\tau_{\ell+1} = \tau_{i_p}$
      $\mathbf{J}^*\{\ell+1\} = \infty$ ▷ There is no valid plan that meets the constraints
      $p = p + 1$ ▷ Increment pivot index
      $b = 1$ ▷ Set *baseline* flag
  else ▷ $p > \ell$
    if $b = 1$ **then**
      $\mathbf{J}^*_{base} = \mathbf{J}^*\{\ell\} - \mathcal{V}^*(\sigma_{i_\ell}, \sigma_*, \tau_{i_\ell})$ ▷ Initialize a *baseline* value for $\mathbf{J}^*\{\ell+1\}$
      $b = 0$ ▷ Reset *baseline* flag
    $\mathbf{J}^*\{\ell+1\} = \mathbf{J}^*_{base} + \mathcal{V}^*_\Diamond(\sigma_{i_\ell}, \sigma_{\ell+1}, \tau_{i_\ell}, \tau_{\ell+1}) + \mathcal{V}^*(\sigma_{i_{\ell+1}}, \sigma_*, \tau_{i_{\ell+1}})$
$\mathbf{J}^*\{\ell+1\}(T_\emptyset) = \mathbf{J}^*\{\ell\}$ ▷ $\tau_{\ell+1}$ is equal to the *null* assignment, namely no constraint

3.2 Plan Merging

The plan merging step of our proposed algorithm requires finding an optimal assignment to the timing variables while optimizing the global cost function $\mathbf{J}^*(\vec{\tau})$. This is a weighted constraint satisfaction programming problem, which is $\mathcal{NP}$-hard in the general case (Larrosa and Dechter 2003). In the special case of the min-sum objective (equation 12), we can reduce optimization complexity by using models that consider the internal structure of the dependency among agents. A graphical model, called factor graph (Loeliger 2004), describes the interaction among agents, and captures agent dependency or independency, therefore leading to more efficient optimization algorithms.

A factor graph contains variable nodes representing constraint variables (the timing variables), and factor nodes representing single-agent cost functions $\mathbf{J}^*_k(\vec{\tau}_k)$. Edges connect a cost function to all the variables associated with the constraints involved in that cost function. Figure 1 illustrates how the min-sum optimization problem is represented using a factor graph. In this example, we have three cooperation constraints, where $\mathbf{G}_1 = \{\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3\}$, $\mathbf{G}_2 = \{\mathbf{g}_1, \mathbf{g}_3\}$, and $\mathbf{G}_3 = \{\mathbf{g}_3, \mathbf{g}_4\}$.

On the factor graph we apply a variable elimination (VE) algorithm, which is used mainly for exact inference (Koller and Friedman 2009). VE exploits the internal structure of the problem and reduces computations (Larrosa and Dechter 2003).

The factor graph structure and the VE elimination ordering have a major effect on the complexity and efficiency of the algorithm, which is out of the scope of this work (see Koller and Friedman 2009). However, in the next section we present several representative cases. The scheme for applying VE to solve the optimization problem is described in (Revach 2018).

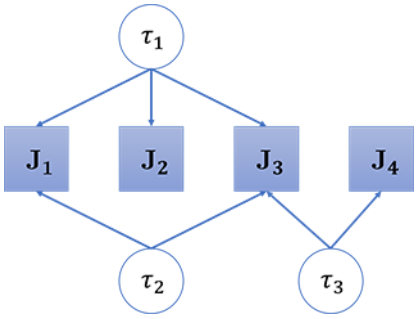


Figure 1: Factor graph example

3.3 Complexity Analysis

In this section we present an overall complexity analysis of our proposed algorithm. We first present the complexity of the response function computation, followed by an overall analysis of a few representative cases, and establish an upper bound on the complexity of planning problems. The complexity result is formulated in terms of the overhead of planning for a multi-agent system as a function of planning for each single agent in isolation, when considering the same problem structure. More specifically, we denote $\mathbf{T}(\mathcal{V}^*, T)$ and $\mathbf{T}(\mathcal{J}^*, T)$ as the time complexity of computing a single-agent cost-to-state and cost-from-state, respectively, over the time horizon T, and assume $\mathbf{T}(\mathcal{V}^*, T) = \mathbf{T}(\mathcal{J}^*, T)$.

For the response function computation, the first step of pre-computation (algorithms 2 and 3) is of the order of $L^2 \cdot \sum_{\tau_\ell \in \mathcal{T}_\ell} \mathbf{T}(\mathcal{V}^*, \tau_\ell) \leq L^2 \cdot \frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*, T)$, where L is the number of constraints in which the agent is involved. We can use an efficient algorithm for computing a single agent cost-to-state from every initial state to a fixed and specific goal state (e.g., using dynamic programming) and denote its time complexity as $\mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T)$. In that case, we may reduce the time complexity by a factor of L, compared to single agent planning, i.e., $L \cdot \frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T)$. The time complexity of the second step (algorithm 4) is dominated by $\mathcal{O}\left(T^L\right)$. Therefore, the overall complexity for computing the response function is

$$L \cdot \frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T) + \mathcal{O}\left(T^L\right) \tag{17}$$

In the case of a single constraint, this reduces to $2 \cdot \mathbf{T}(\mathcal{V}^*, T) + \mathcal{O}(T)$ (equation 15).

The complexity of the plan merging step, and more specifically the VE algorithm, depends on the scope size of each factor; that is, the number of variables to which each factor is connected. The total complexity has an exponential dependency in the scope size of the factors and it is of the order of $\mathcal{O}((K+L) \cdot d^m)$ where m is the maximal scope size of factors and d is the maximal number of values of each variable. For a detailed complexity analysis of the VE algorithm on a factor graph, see (Revach 2018; Koller and Friedman 2009).

We now present an analysis of the overall time complexity for several representative cases. We start with a very sparse case, where there are $2 \cdot L$ agents, each of which is involved in only one cooperation constraint, i.e., $K = 2 \cdot L$. The response function computation time complexity is dominated by $2 \cdot \mathbf{T}(\mathcal{V}^*, T) + \mathcal{O}(T)$ and is linear in the span of the time horizon. Each timing variable does not depend on any of the other variables. The time complexity of eliminating a single variable is dominated by $\mathcal{O}(T)$; i.e., it is also linear in the span of the time horizon. The overall complexity is $K \cdot [2 \cdot \mathbf{T}(\mathcal{V}^*, T) + \mathcal{O}(T)] + L \cdot \mathcal{O}(T) = 2 \cdot K \cdot \mathbf{T}(\mathcal{V}^*, T) + \frac{3}{2} \cdot K \cdot \mathcal{O}(T)$. Because of the inherent decoupling in this case, this is equal to solving $L = \frac{K}{2}$ independent problems.

In a dense case we consider two agents with $L \geq 2$ cooperation constraints between them (i.e., each agent is involved in L constraints). The time complexity of the response function computation is exponential in L. As all the timing variables belong to the same factors, they are therefore dependent. The time complexity of the plan merging is also exponential in L, but it is not the dominating part. The overall complexity is dominated by $2 \cdot L \cdot \left(\frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T) + \mathcal{O}\left(T^L\right)\right)$.

We now consider an hierarchical case, where each constraint involves two agents and the factor graph is a balanced N-tree with depth M. There are N^M agents (factors) that are represented as leaf nodes in the tree, and $K - N^M$ agents that are not represented as leaf nodes. Here, K is equal to $K = \sum_{m=0}^{M} N^m$; therefore, the total number of cooperation constraints is equal to $L = K - 1$. Each of the leaf agents is involved in only one cooperation constraint; therefore, the complexity of computing their response function is just linear: $N^M \cdot (2 \cdot \mathbf{T}(\mathcal{V}^*, T) + \mathcal{O}(T))$. An agent that is not a leaf node in the tree is involved in $N+1$ cooperation constraints. Therefore, the complexity of computing their response function is $\left(K - N^M\right) \cdot (N+1) \cdot \left(\frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T) + \mathcal{O}\left(T^{N+1}\right)\right)$. In the case of a tree, the plan merging is executed bottom up from the leaf nodes to the root node. Every factor that is not a leaf generates an $N+1$ cliques (see Koller and Friedman 2009) of timing variables (i.e., all the variables on which the factor depends). Therefore, the complexity of plan merging is dominated by the size and number of cliques. The complexity of eliminating a clique by a VE algorithm is dominated by $\mathcal{O}\left(T^{N+1}\right)$, and the number of cliques is equal to $\mathbf{C_L} = K - N^M$. Note that the process of eliminating cliques in the same level of the tree can be distributed and parallelized.

Finally, we define a coupling measure ρ for the system as the maximal number of constraints in which each agent is involved,

$$\rho = \max_k L_k, \quad k = 1, \ldots, K \tag{18}$$

where L_k is the number of constraints in which agent $\mathbf{g}_k$ is involved. An upper bound for the complexity is linear in K, polynomial in T, and exponential only in ρ:

$$\mathcal{O}\left(K \cdot \rho \cdot \left(\frac{T}{2} \cdot \mathbf{T}(\mathcal{V}^*_{\mathcal{B}}, T) + T^{\rho}\right)\right) \tag{19}$$

4 Experiments

In this section we present the results of basic experiments performed using DIPLOMA, in order to validate its correctness and test its time complexity. We compare the algorithm's performance against a centralized DP algorithm solving the underlying MMDP. We use the same DP algorithm for calculating $\mathcal{J}^*(\sigma_I, \sigma, \tau)$ and $\mathcal{V}^*(\sigma, \sigma_*, \tau)$ in algorithm 2. All simulations were performed on an Intel i7-8700 CPU @ 3.20Ghz machine with 16.0 GB RAM.

We ran our simulations on a simple grid world example where several agents have to travel from an initial location to a goal location in T time steps while collecting as many boxes as possible. Each box has its own reward and associated agent, and some boxes can be picked by two agents together in order to gain a double reward. In order for agents to pick a box together, they have to meet at the box location at the same time. Agents can move *up*, *left*, or *right*, and

collect the reward upon moving up from their box location. Our goal is to find an optimal joint plan such that the group reward is maximized. Note that in this example we use reward instead of cost used in the model; however, replacing between the two is trivial by taking negative rewards. This problem is depicted in figure 2 for a grid of 10×10 and four agents ($K = 4$). This problem is quite simple but can represent scheduling problems, box-pushing, search-and-rescue and more.

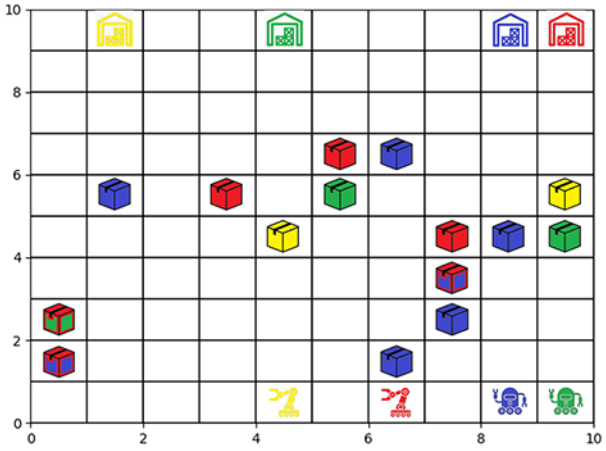


Figure 2: A box collecting problem on a 10×10 grid with four agents ($K = 4$), denoted by four different colors. All agents start in the bottom row and have to arrive to the corresponding warehouse at the top row within T time steps. Each agent can only pick boxes of its color. Agents can cooperate in three different locations ($L = 3$), illustrated by a two-colored box. For instance, in the box located in $(0, 2)$, the red agent can pick the box alone and receive a reward of 3, or it can pick it with the assistance of the blue agent and receive a reward of 6.

We ran our simulation on a fixed grid size of 10×10 ($|\mathcal{S}| = 100^K$), a fixed horizon of $T = 20$ time steps, and different values for number of agents (K), and constraints (L). For every value of K we generated 20 random environments (with a random number of constraints) and measured the runtime of the centralized reference algorithm and DIPLOMA. Figure 3 demonstrates how our proposed algorithm scales in the number of agents, and depends on the coupling measure ρ (see equation 18). We also compare the runtime of our algorithm using the VE algorithm for the plan merging step, compared to a brute-force (BF) optimization. Elimination ordering for VE was determined by a simplified min-neighbors criteria (Koller and Friedman 2009). The reference algorithm does not depend on the coupling measure (i.e., number of constraints), but for $K = 3$, has a runtime higher by two orders of magnitude than DIPLOMA. A value of $K = 4$ makes it practically infeasible to run. DIPLOMA, on the other hand, scales well in the number of agents and depends mostly on the coupling level. Furthermore, using VE optimization for the plan merging step (compared to a brute-force optimization), reduces runtime significantly when the coupling measure increases.

5 Extension to Asymmetric Interactions

In this paper we focus on simple symmetric interactions between agents, i.e., meeting constraints where all agents must arrive at the same time for the group to benefit from the interaction. Our model, however, can be extended to include asymmetric and more complex temporal constraints, enabling a compact representation of such constraints. Furthermore, it enables the development of efficient planning algorithms that exploit the linear time complexity of solving an MDP. This can be done by applying the group interaction cost $\mathcal{C}_\ell^-$ to a specific interacting agent, called the *affected* agent, and embedding an *activation function* of the form $f_\ell : \mathcal{T} \times \mathcal{T}_\ell \rightarrow \{0, 1\}$ into the affected agent's immediate cost. The activation function defines a set of time instances where the interaction cost is applicable.

As an example, one can consider a scenario where a *facilitating* agent can arrive at a certain state in time $\tau \in \mathcal{T} = \{1, ..., 10\}$, which opens a 10 time steps window following time τ, allowing the second agent to receive an additional reward for each time step (within this time window) in which it is in a related state. If we formulate this interaction using a distinct constraint for each possible state-time pair, we need $10^2 = 100$ constraints, and thus checking about $2^{100} \approx 10^{30}$ different combinations of constraints. By formulating this interaction with an SCC, using a step activation function, we would have only 10 constraints. We would need to check only $2^{10} = 1024 \approx 10^3$ combinations of constraints of the first agent, and for each one, solve a single induced MDP for the second agent. Thus, we obtain an improvement of many orders of magnitude in this simple case.

A detailed formulation and implementation of this extension is presented in (Revach 2018), including an efficient asymmetric planning algorithm using a step activation function.

Another rather trivial extension to asymmetric interactions is to use a different interaction timing for every interacting agent in a constraint. Since we calculate the agents' response for each $\tau \in \{0, \ldots, \mathcal{T}\}$ (see algorithm 5), we can choose a different value of τ for each agent in the plan merging step.

6 Discussion and Future Work

In this paper, we address the problem of fully cooperative multiple agents high-level planning problems in deterministic environments. We focus on problems where interactions between agents are symmetric and sparse, and present a framework for representing all interactions as *soft cooperation constraints* (SCC). This framework enables a compact representation of temporal constraints and can be further extended and generalized to include more types of constraints. Considering the SCC formulation, only those agents that are subject to the same cooperation constraint are coupled, forming a dependency only in a specific context.

The SCC model presented is quite general and useful in practice, and can express constraints used in realistic scenarios. The main use case is coordination of high-level actions

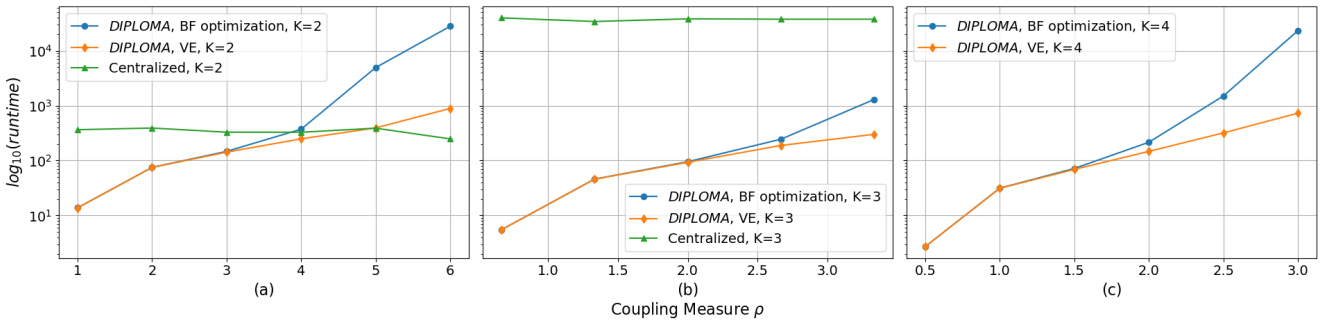


Figure 3: Simulation results for $K = 2$ (a), $K = 3$ (b), and $K = 4$ (c) on a logarithmic scale. The centralized reference planner does not depend on the coupling measure (i.e., number of constraints in the problem), but scales poorly on the number of agents, and for $K = 4$ is practically infeasible. DIPLOMA algorithm achieves an improvement of 2 orders of magnitude, and depends on the coupling level.

among autonomous agents. Example problems are the coordination of rescue or military forces, the Mars rover exploration (discussed in Becker et al. 2004) or the coordinated target tracking (discussed in Kumar and Zilberstein 2009). We can extend several combinatorial optimization problems, such as the vehicle routing problem (VRP) or the multiple traveling salesman problem, to include potential meetings between agents that provide additional rewards for the group. An SCC can also express a conflict (or collision) constraint (specifically in multi-agent path finding problems) by setting a positive or infinite interaction cost (see section 2), and using the extended formulation presented in section 5. In a similar way, we can also represent resource constraints, such as "use at most 1 of this resource at the same time", by adding constraints to states where the resource is used by agents.

Using this model, we are able to describe an efficient algorithm, *DIPLOMA*, which is both complete and optimal. The proposed algorithm is a two-step algorithm: a dynamic programming-based planning step and an optimization step.

In the first step, each agent plans independently and computes its response function to the associated constraints with respect to interaction timing variables. We show non-trivial and efficient algorithms for computation, which can also be distributed and parallelized. The time complexity per agent strongly depends on the span of the time horizon and the number of cooperation constraints relevant to this particular agent.

In the second step, we use a variable elimination algorithm on a factor graph to find the optimal assignment to timing variables. The algorithm exploits the internal structure of the problem and independence among agents to efficiently solve the min-sum optimization problem.

A theoretical time complexity analysis is presented, showing that the overall algorithm is linear rather than exponential in the number of agents, polynomial in the span of the time horizon, and dependent on the number of interactions among agents.

Simulations show that the algorithm is efficient compared to a standard solution and scales well in the number of agents.

An immediate direction for future research is the extension to more expressive interaction constraints, as discussed in section 5. Other possible directions for future research include generalizing the formulation of constraints by expanding the state and time domains of each constraint, defining types of agents (rather than specific agents) in a constraint, approximate methods for computing the response functions, and simulating a real-world large-scale MAP problem.

7 Acknowledgements

We thank our two anonymous reviewers for their useful and constructive comments.

References

Becker, R.; Zilberstein, S.; Lesser, V.; and Goldman, C. V. 2004. Solving transition independent decentralized markov decision processes. *Journal of Artificial Intelligence Research* 22:423–455.

Boutilier, C. 1996. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, 195–210. Morgan Kaufmann Publishers Inc.

Brafman, R. I., and Domshlak, C. 2008. From one to many: planning for loosely coupled multi-agent systems. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 28–35.

Fioretto, F.; Pontelli, E.; and Yeoh, W. 2018. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research* 61:623–698.

Guestrin, C.; Koller, D.; and Parr, R. 2002. Multiagent planning with factored mdps. In *Advances in neural information processing systems*, 1523–1530.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA: MIT Press.

Kumar, A., and Zilberstein, S. 2009. Constraint-based dynamic programming for decentralized POMDPs with structured interactions. In *Proceedings of the International Joint*

Conference on Autonomous Agents and Multiagent Systems, AAMAS.

Larrosa, J., and Dechter, R. 2003. Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints* 8(3):303–326.

Loeliger, H.-A. 2004. An introduction to factor graphs. *IEEE Signal Processing Magazine* 21(1):28–41.

Melo, F. S., and Veloso, M. 2011. Decentralized MDPs with sparse interactions. *Artificial Intelligence*.

Nair, R.; Varakantham, P.; Tambe, M.; and Yokoo, M. 2005. Networked distributed pomdps: A synthesis of distributed constraint optimization and pomdps. In *AAAI*, volume 5, 133–139.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*.

Oliehoek, F. A.; Spaan, M. T.; Vlassis, N.; and Whiteson, S. 2008. Exploiting locality of interaction in factored dec-pomdps. In *Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, 517–524.

Oliehoek, F. A.; Whiteson, S.; and Spaan, M. T. 2012. Exploiting structure in cooperative bayesian games. In *Uncertainty in Artificial Intelligence - Proceedings of the 28th Conference, UAI 2012*.

Revach, G. 2018. Planning for cooperative multiple agents with sparse interactions. Master's thesis, Technion - Israel Institute of Technology, Haifa, IL.

Scharpff, J.; Roijers, D. M.; Oliehoek, F. A.; Spaan, M. T.; de Weerdt, M. M.; et al. 2016. Solving transition-independent multi-agent mdps with sparse interactions. In *AAAI*, 3174–3180.

A Factored Approach To Solving Dec-POMDPs

Eliran Abdoo, Ronen I Brafman, Guy Shani
Ben-Gurion University, Israel
eliranb,brafman,shanigu@bgu.ac.il

Abstract

Dec-POMDPs model planning problems under uncertainty and partial observability for a distributed team of cooperating agents planning together but executing their plans in a distributed manner. This problem is very challenging computationally (NEXP-Time Complete) and consequently, exact methods have difficulty scaling up. In this paper we present a heuristic approach for solving certain instances of Factored Dec-POMDP. Our approach reduces the joint planning problem to multiple single agent POMDP planning problems. First, we solve a centralized version of the Dec-POMDP, which we call the team problem, where agents have a shared belief state. Then, each agent individually plans to execute its part of the team plan. Finally, the different solutions are aligned to achieve synchronization. Using this approach we are able to solve larger Dec-POMDP problems, limited mainly by the abilities of the underlying POMDP solver.

1 Introduction

Decentralized Partially Observable Markov Decision Processes (Dec-POMDPs) are a popular model for planning in stochastic environments under uncertainty with partial observability by a distributed team of agents (Oliehoek and Amato 2016). In this model, a team of agents attempts to maximize the team's cumulative reward where each agent has only partial information about the state of the system during execution. The team can plan together centrally prior to acting, but during execution each agent is aware of its own observations only. Communication is possible only through explicit communication actions, if these are available.

To achieve their common goal agents must coordinate their actions in two ways: First, as in single agent problems, actions must be coordinated sequentially. That is, current actions must help steer the system later towards states in which greater reward will be possible. For example, to be rewarded for shipping a product, it must first be assembled. Second, agents may need to coordinate their simultaneous actions because their effects are dependent, e.g., a heavy box can only be pushed if two agents push it simultaneously.

Our focus is on centralized off-line planning for distributed execution. That is, offline, a solver with access to the complete model must generate a policy for each agent. An agent's policy specifies which action it executes as a function of the agent's history of actions and observations. Such policies can be represented by a *policy graph* where nodes are labeled by actions, and edges are labeled by observations. Online, each agent executes its own policy independently of the other agents. The challenge is to generate policies that provide sufficient coordination, even though each agent makes different observations at run-time. Thus, agents' beliefs over which states are possible are typically different.

Dec-POMDPs are notoriously hard to solve – they are NEXP-Time hard (Bernstein, Zilberstein, and Immerman 2013), implying that only the smallest toy problems are optimally solvable. However, many approximate methods for solving Dec-POMDPs have been proposed, with steady progress. Some of these methods generate solutions with bounds on their optimality (Oliehoek et al. 2013; Seuken and Zilberstein 2007; Oliehoek, Kooij, and Vlassis 2008), and some are heuristic in nature (Nair et al. 2003). However, current methods typically do not scale to state spaces with more than a few hundreds of states.

In this paper we describe a heuristic approach for solving Dec-POMDPs that scales to much larger state spaces. The key idea is to solve a Dec-POMDP by solving multiple POMDPs. First, we solve the *team POMDP*, a POMDP in which every observation by one agent is immediately available to the other agents. Hence, all agents have the same belief state. The solution of the team POMDP can be represented by a policy graph — the *team policy graph*. It provides us with a skeleton for the solution of the Dec-POMDP, specifying what each agent needs to provide for the team. Naturally, this policy is not executable by the agents, because agents cannot condition their actions on the observations of other agents in the real world.

Hence, in the next stage, we let each agent solve a POMDP in which it is rewarded for behaving following the specification in the team policy. This leads to the generation of a policy tree for each agent. These policy trees are often not well synchronized. In the last step we synchronize the policy trees by delaying the actions of agents to improve the probability of good coordination.

We implemented our algorithm and tested it on several configurations of a benchmark problem: Collaborative Box-Pushing which is a variation of the Cooperative Box Pushing

problem. We show that the algorithm manages to scale well beyond current Dec-POMDP solvers. One of the main properties of the domain, is that agents policies are only loosely coupled. That is, the need for actions that affect state components that are relevant to all agent, is sparse. That sparsity allows for each agent to independently construct a plan that consists mostly of its own private actions without requiring it to consider the other agents' behavior. This allows us to achieve good decentralized policies even when achieving the goal requires many steps, compared to planning directly over the Dec-POMDP model.

2 Background

We now provide needed background on POMDPs, Dec-POMDPs, their factored representation, and policies. We also introduce the concept of private and public variables and actions in Dec-POMDPs.

2.1 POMDPs

A POMDP is a model for single-agent sequential decision making under uncertainty and partial observability. Formally, it is a tuple $P = \langle S, A, T, R, \Omega, O, \gamma, h, b_0 \rangle$, where:

- S is the set of states. The future is independent of the past, given the current state.
- A is the set of actions. An action may modify the state and/or provide information about the current state.
- $T : S \times A \rightarrow \prod(S)$ is the state transition function. $T(s, a, s')$ is the probability of transitioning to s' when applying a in s.
- $R : S \times A \times S \rightarrow \mathbb{R}$ is the immediate reward function. $R(s, a, s')$ is the reward obtained after performing a in s and reaching s'.
- Ω is the set of observations. An observation is obtained by the agent following an action, and provides some information about the world.
- $O : S \times A \rightarrow \prod(\Omega)$ is the observation function, specifying the likelihood of sensing a specific observation following an action. $O(s', a, o)$ is the probability of observing $o \in \Omega$ when performing a and *reaching* s'.
- $\gamma \in (0, 1)$ is the discount factor, quantifying the relative importance of immediate rewards vs. future rewards.
- h is the planning horizon — the amount of actions that an agent executes before terminating. The horizon may be infinite.
- $b_0 \in \prod(S)$ is a distribution over S specifying the probability distribution over the initial state.

For ease of representation, we assume that agent actions are either sensing actions or non-sensing actions. An agent that applies a non-sensing action receives the observation *null-obs*. In addition, we also assume that every action has an effect that we consider as the successful outcome, while all other effects are considered failures. We later explain how this assumption can be omitted in the relevant parts.

Often, the state space S is structured, i.e., it consists of assignments to some set of variables $X_1, \ldots X_k$, and the observation space Ω is also structured, consisting of a set of observation variables $W_1, \ldots, W_d$. Thus, $S = Dom(X_1) \times \cdots \times Dom(X_k)$ and $\Omega = Dom(W_1) \times \cdots \times Dom(W_d)$. In that case, τ, O, and R can be represented compactly by, e.g., a dynamic Bayesian network (Boutilier, Dean, and Hanks 1999). Formats such as RDDL (Sanner 2011) and POMDPX (POM 2014) exploit factored representations to specify POMDPs compactly.

Example 1. Consider a simple Box-Pushing in a 2 cell grid. The left cell is marked by *L* and the right cell by *R*. The agent begins in the left cell. There is a single box, that starts in the right cell. The agent can either move, sense its current cell or push a box from its current cell. Both move and push can be done in any direction — left and right. The agent's goal is to push the box to the right cell. The state is composed of 2 variables: the location of the agent and the location of the box. Each variable can take one of two values: *L* or *R*. The sense action returns an observation telling whether there's a box in the agent's cell, while the move and push actions are non-sensing actions always returning *null-obs*. The push action has a success probability of 0.8.

A solution to a POMDP can be formed as a *policy*, assigning to each history of actions and observation (*AO-history*) the next action to execute. Such a policy is often represented using a *policy tree* or, more generally, a *policy graph* (also called a finite-state controller).

A policy graph $G = (V, E)$ is a directed simple graph, in which each vertex is associated with an action, and each edge is associated with an observation. For every edge $v \in V$ and every observation $o \in \Omega$ exactly one edge emanates from v with the label o. The graph has a single root which acts as its entry point. Every AO-history h can be associated with some path from the root to some vertex v, and the action labelling v is the action that the policy associates with h.

Finally, using a policy graph to direct the agent on the problem produces a *trace* – an execution trajectory. A trace T of length l is a sequence of quintuplets $e_i = (s_i, a_i, s'_i, o_i, r_i)$, namely *steps*, that occurred during a possible policy execution where: s_i is a state in step i and s_0 is the initial state; a_i is the action taken in step i; s'_i is the result of applying a_i in s_i; o_i is the observation received after taking a_i and reaching s'_i; and is the reward received for taking the a_i in s_i and reaching s'_i. Clearly, $\forall i$ such that $0 \leq i \leq l - 1$, we have $s'_i = s_{i+1}$.

2.2 Dec-POMDP

A Dec-POMDP models problems where there are $n > 1$ acting agents.. These agents are part of a team, sharing the same reward, but they act in a distributed manner, sensing different observations. Thus, their information state is often different. Formally, a Dec-POMDP for n agents is a tuple $P = (S, A = \bigcup_{i=1}^{n} \{A_i\}, T, R, \Omega = \bigcup_{i=1}^{n} \{\Omega_i\}, O, \gamma, h, \{I_i\}_{i=1}^{n})$, where:

- S, γ, h, b_0 are defined as in a POMDP.
- A_i is the set of actions available to agent i. We assume that A_i contains a special *no-op* action, which does not change the state of the world, and does not provide any informative observation. $A = A_1 \times A_2 \times .. \times A_n$ is the

set of joint actions. On every step each agent i chooses an action $a_i \in A_i$ to execute, and all agents execute their actions jointly. $\langle a_1, ..., a_n \rangle$ is known as a joint action. We often treat the single-agent action a_i as a joint-action, with the understanding that it refers to the joint-action $\langle$no-op,. . . , a_i, . . . ,no-op$\rangle$

- $T : S \times A \to \prod(S)$ is the transition function. Transitions are specified for joint actions, that is, $T(s, \langle a_1, ..., a_n \rangle, s')$ is the probability of transitioning from state s to state s' when each agent i executes action a_i.
- $R : S \times A \times S \to \mathbb{R}$ is the reward function. Rewards are also specified over joint actions.
- $\Omega = \Omega_1 \times \Omega_2 \times .. \times \Omega_n$ is the set of joint observations. Each Ω_i contains a special *null-obs* observation received when applying a non-sensing action.
- $O : S \times A \to \prod_{i=1..n}(\Omega_i)$ is the observation function, specified over joint actions. $O(s', \langle a_1, ..., a_n \rangle, \langle o_1, ..., o_n \rangle)$ is the probability that when all agents execute $\langle a_1, ..., a_n \rangle$ jointly and reach s', each agent i observes o_i.
- γ is the discount factor.
- h is the horizon.
- $b_0 \in \prod(S)$ is a distribution over S specifying the probability that each agent begin its execution in each state. In principle, different agents may have different initial belief states, but we make the (common) assumption that the initial belief state is identical.

Example 2. We extend the previous example to a Dec-POMDP by adding an agent at the right cell and a second box that starts in the left cell. The agents are denoted by *Agent1* and *Agent2* and the boxes by *Box1*, and *Box2*. *Box1* must reach the left cell, and *Box2* must reach the right cell.

As in the case of POMDPs, Dec-POMDPs can also be represented in a factored manner (Oliehoek et al. 2008), although most work to date uses the flat-state representation. We add the notion of *observation variables*, which capture the observation value each agent obtains following an action. Each observation variable is denoted by ω_i which takes values in Ω_i, and represents the observation of agent i.

Example 3. In our example, the state is now composed of 4 state variables: the location of each box – (X_{B1}, X_{B2}) – and the location of each agent – (X_{A1}, X_{A2}). In addition, there are two observation variables – (ω_1, ω_2).

An important element of a factored specification of Dec-POMDPs is a compact formalism for specifying joint-actions. If there are $|A|$ actions in the domain, then, in principle, there are $O(|A|^n)$ possible joint actions. Specifying all joint actions explicitly is unrealistic for large domains.

In many problems of interest we may expect that most actions will not interact with each other. A pair of actions $a \in A_i$, $a' \in A_j$ is said to be non interacting, if their effects when applied jointly (in the same joint action) is the union of their effects when applied separately. Thus, our specification language focuses on specifying the effects of single-agent actions and specific combinations of single-agent actions that interact with each other, which we refer to as *collaborative* actions (Bazinin and Shani 2018). For a more detailed discussion of the compact specification of joint-actions, see (Shekhar and Brafman 2020).

Example 4. We alter our example further by introducing a collaborative action. To do so, we need to convert one of the boxes to a "heavy" box - a box that requires both agents to push it. We will convert *Box1* to such box. Both agents now have also the option to apply a *collaborative-push* action in any specified direction. If both agents apply that action to push *Box1* while in the same cell with it, the box will transit.

Finally, a solution to a Dec-POMDP is a set of policies ρ_i, one for each agent. It maps action-observation sequences of this agent to actions in A_i. As in POMDPs, these policies can be specified using a policy graph for each agent. The policy graph for agent i associates nodes with actions in A_i and edges with observations in Ω_i.

2.3 Public and Private Actions and Variables

We find it useful to define the concept of public and private variables and actions. State variables can influence or be influenced by an agent's action. State variables that are influenced by several agents are called *public* and state variables that are influenced only by a single agent are *private*. The concept of private and public (or *local* and *global*) variables (Brafman and Domshlak 2006) has been used extensively in work on privacy-preserving multi-agent planning (e.g., (Nissim and Brafman 2014; Maliah, Brafman, and Shani 2017)) and, more recently, in work on solving qualitative variants of Dec-POMDPs (Brafman, Shani, and Zilberstein 2013; Shekhar, Brafman, and Shani 2019).

We now explain how we extend these concepts to factored Dec-POMDPs. These definitions are based on the notions of preconditions and effects, as used in classical planning. Let $a \in A_i$ be a non-sensing action of agent i. We identify a with the joint action (no-op, . . . , a, . . . , no-op). We say that a state variable X_i is an *effect* of a if there is some state s for which there is a positive probability that the value X_i changes following a. We denote the effects of a by $eff(a)$.

We say that state variable X_i is an *influencer* of a if a behaves differently for different values of X_i. That is, if there are two states s_1, s_2 that differ only in the value of X_i such that $R(s_1, a, s') \neq R(s_2, a, s')$, or $T(s_1, a, s') \neq T(s_2, a, s')$ for some state s', or $O(s_1, a, o) \neq O(s_2, a, o)$ for some observation o. We denote influencers of a by $inf(a)$. We refer to the union of the influencers and effects of a as the *relevant* variables of a, denoted $rel(a)$.

If a is a sensing action, we define $inf(a)$ similarly, i.e., $X_i \in inf(a)$ if there are two states s_1, s_2 that differ only in the value of X_i such that the distribution over the values of the observation variable of the agent performing a at s_1 and at s_2 are different.

For collaborative actions, the definitions remain the same except that now we identify a with the the joint-action that is composed of the actions of the collaborating agents, and *no-ops* for the rest.

We say that a variable X_i is *relevant* to agent j, if X_i is relevant to some some $a \in A_j$. Finally, X_i is *public* if it is relevant to more than one agent, and it is private otherwise.

Example 5. In our running example, X_{B1} and X_{B2} are both public variables, as they are relevant of both agents' push actions. X_{A1}, X_{A2} are private variables of agent 1 and agent 2 respectively, as they are the relevant only to each respective agent's move action. The same holds for ω_1 and ω_2 with respect to the sense actions. Furthermore, the move and sense actions are private actions, while the push actions are public.

3 FDMAP - Factorized Distributed MAP

Given the input factored Dec-POMDP problem P, we first generate the team POMDP P_{team}. We solve P_{team} using an off-the-shelf POMDP solver – we used SARSOP (Kurniawati, Hsu, and Lee 2008) – and output the team policy.

Next, we then use the team policy to produce traces, which are simulations of the team policy over the team problem. Using the traces, we project the team problem with respect to each agent, as follows: First, for each agent, we extract from the traces a set of public actions and the context in which they were applied, which we call a *contexted actions*. The context captures the conditions under which the action achieves the same effects as in the trace. Then, we associate a reward with each contexted action. The reward associated with the contexted action is designed so that agents will be rewarded for acting in a manner similar to their behavior in the team solution.

Using these contexted actions and their rewards, together with the factored Dec-POMDP, we generate one single-agent problem for each agent. The dynamics of each single-agent problem is similar to that of the Dec-POMDP, except that some variables are projected away.

Finally, we process the single-agent policies and align them to try and ensure that actions are properly synchronized when they are executed in a decentralized manner. In the rest of this section we explain the steps that follow the generation of the team solution in more detail.

3.1 Producing the Traces

Having generated the team problem, P_{team}, we solve it and produce the traces, to capture the possible scenarios of the problem. We must specify three hyper-parameters: a pair of confidence parameters α, β and a precision parameter ϵ_{team}. We generate an ϵ_{team}-optimal solution to P_{team} using an off-the-shelf POMDP solver, and then simulate that solution to produce n_t traces. We want the the empirical distribution of the initial states observed in the traces and the distribution given by the initial belief state to be close. To do so, we generate sufficiently many traces so that the probability of the KL-Divergence to be greater than β, is less than α.

To pick the number of traces we use a result on concentraion bounds for multinomial distribution from (Agrawal 2019), since the initial belief state b_0 is a multinomial distribtuion. We denote by T_0 the sampled distribution, and by k the number of initial states, namely the support set of b_0. Using the theorem, for every $n_t > \frac{k-1}{\beta}$ we have:

$$Pr(KL(T_0||b_0) \geq \beta) \leq e^{-n_t \cdot \beta} \left(\frac{e\beta n_t}{k-1} \right)^{k-1}$$

Example 6. In our example, solving the team problem yields the following policy graph. *Agent1* starts by pushing *Box2* to the right, and then senses whether it had succeeded. It then moves left to assist *Agent2* to push the heavy box, *Box1*, to the right, and again senses to verify its success.

Next, we use the policy graph to produce the traces. Different traces will differ by the number of pushes the agents performs until success. Table 1 shows two possible traces. Recall that the state is composed of 4 state variables: $(X_{A1}, X_{A2}, X_{B1}, X_{B2})$, where each variables can take values in *(L, R)*. The actions' names will be denoted by the action name (M for move, P for push, CP for collaborative push and S for sense), followed by the direction for move and push actions (L, R), and sub-scripted by the target box for sense and push actions ($B1$, $B2$). We also denote the null-obs by ϕ

	X_{A1}	X_{A2}	X_{B1}	X_{B2}	a_1	a_2	ω_1	ω_2
1	L	R	R	L	PR_{B2}	IDLE	ϕ	ϕ
2	L	R	R	R	S_{B2}	IDLE	ϕ	no
3	L	R	R	R	MR	IDLE	ϕ	ϕ
4	R	R	R	R	CPL_{B1}	CPL_{B1}	ϕ	ϕ
5	R	R	L	R	S_{B1}	IDLE	no	ϕ
1	L	R	R	L	PR_{B2}	IDLE	ϕ	ϕ
2	L	R	R	R	S_{B2}	IDLE	ϕ	no
3	L	R	R	R	MR	IDLE	ϕ	ϕ
4	R	R	R	R	CPL_{B1}	CPL_{B1}	ϕ	ϕ
5	R	R	R	R	S_{B1}	IDLE	yes	ϕ
6	R	R	R	R	CPR_{B1}	CPL_{B1}	ϕ	ϕ
7	R	R	L	R	S_{B1}	IDLE	no	ϕ

Table 1: An example of two traces

3.2 Extracting Contexted Actions

We seek a policy for each agent in which the agent's public actions executions are identical to those that appear in the team plan. That is, the agent should execute the same public actions it executes in the team plan and in the same contexts. To generate such a policy, we define an appropriate reward function for each agent that encourages the agent to execute the public actions in the team plan in its own plan and in the same context.

The context of an action must capture the conditions under which the policy chooses the specific action to be executed. We can associate the context with a specific state, but this is too restrictive, as the state might contain various variables that are irrelevant to the action. It is preferable to define a less restrictive context that generalizes to all states where the action achieves the same effects.

Definition 1. The *context* of an action a for agent i is the set of values $\langle x_{j_1}, ..., x_{j_k} \rangle$ for the public variables and the private variables of agent i.

Definition 2. A *contexted action* (CA for short) is a pair $\langle c, a \rangle$, where a is a public action and c is the context of a, such that there exists a trace t and an index i, where $t_i = \langle s, a, \omega \rangle$, and c and s assign identical values to the context variables of a for agent i.

We extract the CAs for agent i, denoted *CActions*$_i$ from the traces. For each trace t, we identify all the public actions in t. For each such public action a of agent i executed in a state s, we identify the context c — the values that s assigns to the *public* variables of the problem and private variables of agent i. In many cases, even though an action a is executed in two different states s_1, s_2 in the traces, the context is identical, as we are interested only in the values of the state variables that are relevant to the execution. We focus on the *public* actions because the projected single agent problems are designed to plan execute these actions only in their appropriate context.

Example 7. Returning to our Box-Pushing example: we find the following public CAs of *Agent1* in the traces:

- *(L,R,R,L),*PR_{B2}
- *(R,R,R,R),*CPL_{B1}

We construct *CActions*$_1$ for *Agent1* by taking the values of the public variables of the problem, and the private variables of *Agent1*. The public variables are X_{B1}, X_{B2}, while the private variable of *Agent1* is X_{A1}.

This results in the following CAs:

- $\langle X_{A1} = L, X_{B1} = R, X_{B2} = L\rangle, PR_{B2}$
- $\langle X_{A1} = R, X_{B1} = R, X_{B2} = R\rangle, CPL_{B1}$

We next describe how the single-agent problems are constructed, given the *CActions*$_i$ sets.

3.3 Single Agent Projection

Our next step is to define a factored POMDP P_i for each agent i. P_i is designed to incentivize the agents to execute the contexted actions of i in the appropriate context. The actions of other agents are used to "simulate" some of the behaviors of the other agents – behaviors that eventually enable the agent to carry out its own actions. P_i contains all state variables of the original problem. It also contains actions of agent i and of other agents. Other agents' actions are included to allow i to simulate their behavior. More specifically, P_i contains all and only the public actions that appeared in some trace of the team plan. In addition, P_i contains all sensing actions of i, but not those of other agents.

For each private action a of another agent, P_i contains a deterministic version of a. Here, we use the assumption that each action has a known desired effect, and the deterministic version of a always achieves this desired effect. This determinization is done mainly to simplify P_i, allowing us to scale to larger problems. We can avoid this determinization, at the cost of a more complicated P_i.

We now design the reward function of P_i, R'_i. The rewards incentivize the execution of the CAs in their appropriate context, while the penalties discourage the agent from applying them outside their context, so that its policy would emulate its behavior in the team plan.

When considering the single agent problems, we no longer want to reward the agents for achieving the team problem goals, but rather reward them for achieving their own parts of the team solution. Therefore, to make the new goals beneficial, we need their associated rewards to surpass the cost required to achieve them. To do so, we take a heuristic upper bounding approach, which manifests the following idea: In the single agent solutions, each CA will be preceded by a sequence of private actions. If the reward for applying that CA would surpass the total cost of its preceding actions sequence sequence, the agent will find it beneficial to apply. Furthermore, if satisfied for all CAs, the whole compound of single agent goals would become beneficial to achieve.

Let $MaxCost$ be the maximal negative reward received by a *private* action, that appeared in the traces. Recalling that we assume each action to have a single successful outcome, let $MinSP$ be the minimal success probability of all actions in the problem, and $MinCASP$ the minimal success probability of the actions appearing in the CAs. Let MTL be the maximal trace length we produced, and MPG be the Maximal Public Gap – the maximal number of *private* actions that precede a public action in the traces. We set $\epsilon > 0$ to some small positive value. Let $CA = \langle c, a\rangle$ be a contexted-action and $Cost(CA)$ be the maximal negative reward received from CA in the traces (and 0 if it was always positive). We compute r_{CA}, the reward given to CA:

$$\frac{\sum_{i=MTL-MPG}^{MTL-1}(\frac{MaxCost}{MinSP}\cdot\gamma^{i-1}) + \frac{Cost(CA)\cdot\gamma^{MTL-1}}{MinSP} + \epsilon}{\gamma^{MTL-1}\cdot MinCASP}$$

The numerator is an upper bound on the expected discounted cost we would pay before applying CA as the last CA (the preceding sequence cost + the cost of the CA itself). The denominator amplifies that cost to be beneficial when scheduled as the last action in the policy.

As noted, we penalize the application of *public* actions in contexts other than those they appeared in in the team plan. This also eliminates potential positive reward cycles (Ng, Harada, and Russell 1999) that can cause an agent to endlessly achieve one of its sub-goals. The penalty is chosen to be $-max_{CA\in CActions_i} r_{CA}\cdot |CActions_i|$, as an upper bound on the sum of rewards that can be achieved from applying CAs. This ensures that public actions executed out of context cost more than applying all the CAs. There is no penalty for other agents' CAs (i.e., contexted actions in $\bigcup_{j\neq i} CActions_j$). We want to allow the agent to simulate other agents' CAs in order to plan the execution of its own actions at appropriate times, and also in order to act based on the uncertainty these actions can introduce.

Finally, we remove rewards related to public variables the agent can achieve because single-agent POMDP's role is to imitate the team policy, not compute an alternative solution.

Given an action $a \in A_i$, a source state s and a state $s' \neq s$ which differs from s on at least one variable $X \in \mathit{eff}(a)$, we set $R'_i(s,a,s') = R(s,a,s') - \mathit{max}\,(0, R(s,a,s') - R(s,a,s))$

Example 8. We now construct *Agent1*'s single-agent problem. We denote the CAs from the previous example with ca_1, ca_2, and their reward with r_{ca_1}, r_{ca_2}. We follow the projection stages one by one:

1. As the push action is the only public action in the problem, we remove all push actions except for the ones that are observed in the traces. For *Agent2* we leave only CPL_{B1} and for *Agent1* we leave CPL_{B1} and PR_{B2}.

Notice that we keep *Agent2*'s CPL_{B1} action in *Agent1*'s problem, as we might need to simulate it.

2. We remove the sensing action of *Agent2*, as well as its observation variable.
3. We don't have any non-deterministic private actions, so no actions are turned to deterministic.
4. We add a penalty of $-2 \cdot max(r_{ca_1}, r_{ca_2})$ to the remaining public actions, applied in any context except for the CA's contexts.
5. We add the rewards r_{ca_1}, r_{ca_2} to ca_1 and ca_2 respectively.
6. The rewards for pushing the boxes to the target cells are set to 0 - we reward the agent only for doing its public action in context.

3.4 Policy Adjustment and Alignment

We run the planner on each of the single agent projections that we generate, constructing a set of single agent policy graphs for the projections. We now adapt the policies to apply to the joint problem.

First, the projection of agent i contains private actions of other agents that must be removed. We traverse through the policy graph and replace every action of another agent by its child. As we do not allow sensing actions of other agents, there is always a single child to actions of other agents.

We now align the policies to increase the chance that actions of different agents occur in the same order as in the team plan. An action a_1 of agent 1 that sets the context value of a variable in the context of an action a_2 of agent 2 should be executed before a_2. Also, collaborative actions should be executed at the same time by all participating agents.

Given stochastic action effects, we cannot guarantee that the policies will be well correlated. It is desirable that we will maximize the probability of synchronization, but currently, we only offer a heuristic approach that empirically improves synchronization probability.

For each public action a in the team plan we select a simple path from the root to a. The *identifier* of the action is the sequence of public actions along the simple path. Public actions in individual agent policy graphs that share the same identifier are assumed to be identical in all graphs.

For a public action a that has the same identifier in all agent graphs, let l be the length of the longest simple path to the action in all policy graphs, including private actions. In any graph where the length is less than l, we add *no-op* actions prior to a to delay its execution.

We use an iterative process — we begin with the action with the shortest identifier (breaking ties arbitrarily), and delay its execution where needed using *no-op*s. Then, we move to the next action, and so forth. After the alignment we remove from each agent's aligned policy graph all the *public* actions of other agents.

Finally, we handle the problem of a potential "livelock" between *collaborative* actions. Consider a scenario where two agents need to perform a non-deterministic collaborative action whose effect can be directly sensed. To do so, after executing the action, both agents perform a *sensing* action that senses the effect of that collaborative action. In executions, the agents may be unsynchronized, applying the collaborative and sensing actions in an alternating manner, where one agent performs the collaborative action while the other performs the sensing action, causing them to enter a livelock. To handle that, given a collaborative action with n collaborating agents, we modify the graph so that every collaborative action that is part of a cycle is repeated by every agent for n times instead of just once. This way, a livelock can never occur.

Example 9. Figures 1(a) and 1(b) show *Agent1*'s policy graph before and after the alignment and adjustments procedure. Since *Agent2*'s job is only to collaboratively push *Box1* with *Agent1*, there are no action simulations of *Agent2* in *Agent1*'s policy, hence no alignment is required. The only modification that occurs is the insertion of the live-lock handling.

4 Empirical Evaluation

We provide experimental results focusing on longer planning horizons and scaling up with respect to current Dec-POMDP solvers. The experiments were conducted on a variation of the popular cooperative box pushing problem. We compare our algorithm, FDMAP, with two Dec-POMDP solvers, GMAA-ICE (Oliehoek et al. 2013) and JESP (Nair et al. 2003), using MADP-tools. (Oliehoek et al. 2017). We evaluate FDMAP, GMMA-ICE and JESP on a Linux machine with 4 cores and 8GB of memory.

4.1 Collaborative Box-Pushing

In the cooperative box pushing domain, agents on a grid can move and push boxes in four principle directions, or perform a *no-op*. Light boxes can be pushed by a single agent, while heavy boxes can only be pushed by a collaborative push of two agents. All actions except for the push actions are deterministic.

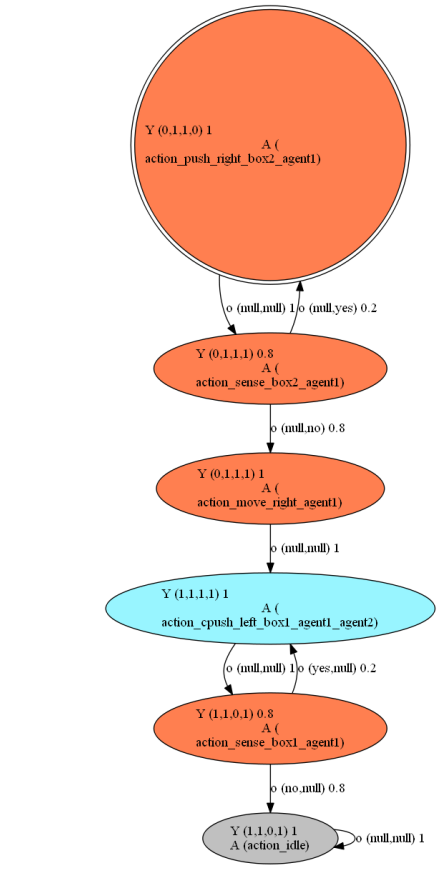
Each grid cell can contain any number of agents and boxes, and each agent can sense for a box at its present location. Initially, each box can appear in either the target cell (and hence, need not be moved) or the lower right cell, with equal probability. The goal of the agents is to move the boxes to a target cell, located at the upper left corner of the grid.

Each action (except for *no-op*) has a cost — 10 for moving, 1 for sensing (encouraging sensing rather than blindly pushing), 30 for pushing, and 20 for a collaborative push (per agent). The reward for moving a box to its target position is 500. In addition, there's a penalty of 10000 for pushing a box *out* of the target cell to avoid abuse. In configurations with heavy boxes we double the reward and penalty. A domain instance of m cells with n agents, l light boxes, and h heavy boxes, has m^{n+l+h} states, $(5 \cdot (l+1) + 4h \cdot (n-1))^n$ actions and 3^n observations.

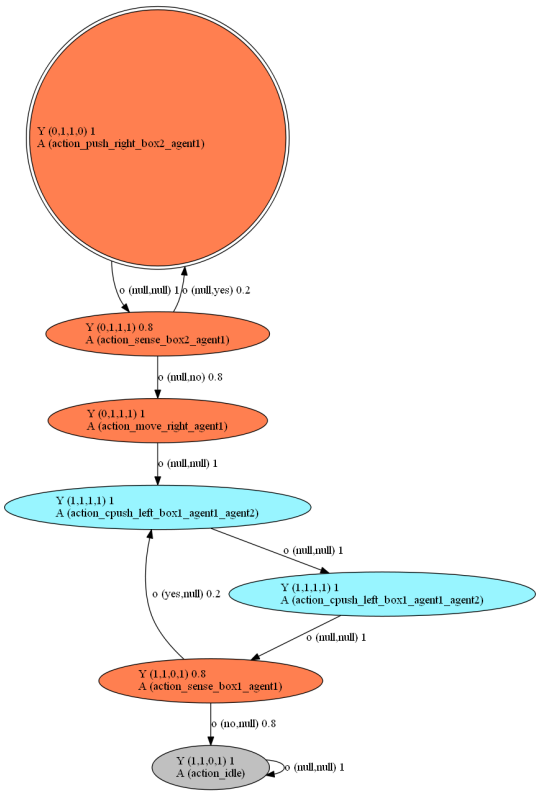
Evidently, the box pushing domain calls for longer horizon policies, rather than good local reactive policies, and requires careful coordination to ensure that collaborative push actions are performed simultaneously by the agents.

4.2 Settings

We compared FDMAP with GMAA-ICE and DP-JESP. GMAA-ICE and DP-JESP require an horizon specification,



(a) Before



(b) After

Figure 1: *Agent1*'s Policy

while FDMAP computes a policy for an unbounded horizon. Therefore, we specify the maximal reached planning horizon for them under the column -H-, and for FDMAP we specify the average number of steps until reaching the goal state, under the column -Avg-. Discount factor is set to $\gamma = 0.99$.

For GMAA-ICE and DP-JESP we report the computed policy value. For FDMAP we measured the average discounted accumulated reward of 1000 simulations.

We provide results on several configurations of Collaborative Box-Pushing. Each box starts at either the top-left or bottom-lower corner with equal probabilities. The problem name convention is composed of 5 marks (each mark specified in brackets), $BP - [w][h][n][l][h]$. The marks stand for Width, Height, Number of agents, Number of light boxes, Number of heavy boxes. For 1 dimensional grids, DP-JESP and GMAA-ICE were given a minimalistic version of the problem that does not include unnecessary actions — push and move for the up and down directions — to decrease the domain size. We specify the agents' initial locations (denoted by I), as well as the domain size, alongside the configuration name.

All planners were given a total of 3600 seconds to solve each $\langle$*configuration, horizon*$\rangle$ pair. In addition, we also limited the solution of each single-agent problem withint FDMAP to 900 seconds. For the hardest problem configuration (BP-33221), we also present results for larger time limit, as 900 seconds were not sufficient for the agents to solve their P_i.

The time shown for DP-JESP and GMAA-ICE is based only on its log from MADP-Toolbox, and does not include the problem loading, which is in many cases non negligible. FDMAP time does not include writing the SARSOP policies and graphs to disk, as they are highly dependent on hardware quality and can effectively remain in memory throughout the whole process.

In both GMAA-ICE and DP-JESP, configurations BP-32302, BP-32303 and BP-33221 could not be solved for any horizon. Therefore we provide comparisons only for the first four configurations (Table 2), while the harder configurations are shown in Table 3. We also provide a more sensitive analysis of the horizon in table 5

In DP-JESP, $\times$ marks a timeout. In GMAA-ICE we mark two different timeout options: FF refers to failure of finding a full policy for the required horizon, where FH refers to an earlier stage timeout when computing the heuristic function.

4.3 Results

The main comparison is presented in Table 2. We can see that FDMAP manages to produce policies with higher quality, as these policies require horizons beyond those that the other planners can handle. We also see that FDMAP's planning time is significanlty smaller. This is due to the fact that FDMAP does not plan directly on the decentralized model, but rather solves multiple POMDP models, which are known to require much less computational effort (Bernstein, Zilberstein, and Immerman 2013).

For the largest problems, shown in Table 3, FDMAP still manages to produce good quality policies, yet with signifi-

DP-JESP			GMAA-ICE			FDMAP		
BP-31211 $\|S\| = 81, \|A\| = 16, I = \langle(1,2),(1,3)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
4	1861.30	279	4	30.23	330.07	15	**2.03**	**590.82**
BP-22202 $\|S\| = 256, \|A\| = 225, I = \langle(1,2),(2,2)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
3	267.24	271	3	160.18	320.46	9	**3.88**	**356.08**
BP-22203 $\|S\| = 1024, \|A\| = 400, I = \langle(1,2),(2,2)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
2	59.06	0	2	1053.27	414	17	**21.01**	**518.02**

Table 2: The results for configurations BP-31211, BP-22202 and BP-22203. FDMAP outperforms DP-JESP and GMAA-ICE with respect to both running time and policy value. The running time is improved significantly. Results for DP-JESP and GMAA-ICE are for maximal horizon reached, specified under the -H- column. In FDMAP we present the average number of steps until reaching the goal state when running the policy for an unbounded number of steps, specified under the -Avg- column

FDMAP					
Problem	MaxSteps	AvgSteps	Time	Value	%Wins
BP-32302 $\|S\| = 7776, \|A\| = 3375$ $I = \langle(1,2),(1,3),(2,3)\rangle$	40	14	92.70	243.91	100
BP-32303 $\|S\| = 46656, \|A\| = 8000$ $I = \langle(1,2),(1,3),(2,3)\rangle$	43	21	1799.94	353.82	98
BP-33221 $\|S\| = 59049, \|A\| = 324$ $I = \langle(1,3),(3,1)\rangle$	124	37	2962.29	8.64	95
BP-33221 3600 seconds per agent	105	35	5379.32	345.56	100

Table 3: Results for the largest scale problems, which only FDMAP managed to solve. Running times are rapidly increasing while reaching the scales limit of the underlying POMDP solver. The policies are still very robust, and reach the goal state in most cases (%Wins). MaxSteps and AvgSteps specify the maximal and average number of steps made until reaching a goal state throughout the runs.

DP-JESP			GMAA-ICE			FDMAP		
BPPEN-31211 $\|S\| = 81, \|A\| = 16, I = \langle(1,2),(1,3)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
3	25.95	0	5	3537.67	438.95	15	**1.99**	**568.20**
BPPEN-22202 $\|S\| = 256, \|A\| = 225, I = \langle(1,2),(2,2)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
3	495.61	135.40	3	446.03	213.79	9	**3.72**	**289.55**
BPPEN-22203 $\|S\| = 1024, \|A\| = 400, I = \langle(1,2),(2,2)\rangle$								
H	Time	Value	H	Time	Value	Avg	Time	Value
2	38.70	0	2	1054.5	326.504	15	**14.67**	**533.26**

Table 4: Results for a variation of Collaborative Box-Pushing, in which we penalize an agent for pushing a light box blindly

BP-21210 $\|S\| = 8, \|A\| = 16, I = \langle(1,1),(1,2)\rangle$						
H	DP-JESP		GMAA-ICE		FDMAP	
	Time	Value	Time	Value	Time	Value
4	19.87	0	**1.15**	**426.91**	1.28	329.34
5	1069.95	0	2.09	438.34	”	321.12
6	×	-	6.97	448.19	”	337.63
7	×	-	8.98	450.97	”	416.74
Max	1069.95	0 (5)	17.1	**454.7** (25)	**1.28**	414.36 (4)

Table 5: Results for BP-21210. FDMAP outputs a reasonable value compared to GMAA-ICE, which optimizes the small scaled problem. The last row presents results for the maximal horizon reached on DP-JESP and GMAA-ICE, and average simulations steps until reaching the goal state for FDMAP when run for unbounded number of steps. These are specified in parentheses next to the policy value

cantly longer running time. The size of the hardest configuration, BP-33221, approaches the maximal problems that SARSOP (Kurniawati, Hsu, and Lee 2008) can solve.

Table 5 shows that FDMAP manages to produce reasonable results compared to the other solvers even when dealing with extremely small domains in which optimal solvers such as GMAA-ICE can excel.

To observe the difference between the policies FDMAP produces to the ones GMAA-ICE does, we present another variation of the domain, where we add a penalty of 101 to light boxes push actions that occur with no box in the cell. The penalty is chosen to be slightly higher than the reward for pushing a box to the target cell, times the fail probability of the push action. Table 4 presents the results (configuration name prefixed with BPPEN). We can see that the the reward difference on maximal results compared to Table 2 are much lower for FDMAP, indicating that FDMAP's policies in the non-penalty configurations exploit the horizon and avoid blindly pushing, leading to higher-quality results. If we would handle domains in which reward can be *only* achieved in large horizons, a case that was mentioned in contexts of reward shaping (Brys et al. 2014; Laud and DeJong 2003), we expect FDMAP's ability to scale to very large horizons while managing to reward agents for their public goals, to become crucial.

5 Conclusion And Future Research

We presented FDMAP — an algorithm for solving factored Dec-POMDPs. FDMAP begins by solving a centralized POMDP, which we call the team POMDP, obtaining a team plan. Then, FDMAP creates agent specific POMDPs whose solutions encourage agents to complete their role in the team plan. The agent plans are then aligned for synchronization between the agents. We experiment with box pushing examples that require collaboration, showing that we can scale to much larger problems than current Dec-POMDP solvers, while computing a reasonable policy.

There are two direction in which we deem FDMAP can be improved, in both scalability and solution quality. The use of online planners instead of SARSOP as the underlying POMDP solver, can greatly improve the scale of solvable

problems. The changes in terms of algorithm's structure are minor, as we merely need to be able to produce the single agent policy graphs using an online solver. In terms of solution quality, we aim at using more principled methods of reward shaping, that come from the field of reinforcement learning in forms of multi-objectivization (Brys et al. 2014). Our goal would be to convert the concept of contexted actions into objectives of each agent, while preserving optimality with respect to the decentralized problem.

Acknowledgements

This work was supported by ISF Grants 1651/19, by the Israel Ministry of Science and Technology Grant 54178, and by the Lynn and William Frankel Center for Computer Science.

References

Agrawal, R. 2019. Concentration of the multinomial in kullback-leibler divergence near the ratio of alphabet and sample sizes. *CoRR* abs/1904.02291.

Bazinin, S., and Shani, G. 2018. Iterative planning for deterministic qdec-pomdps. In Lee, D.; Steen, A.; and Walsh, T., eds., *GCAI-2018. 4th Global Conference on Artificial Intelligence*, volume 55 of *EPiC Series in Computing*, 15–28. EasyChair.

Bernstein, D. S.; Zilberstein, S.; and Immerman, N. 2013. The complexity of decentralized control of markov decision processes. *CoRR* abs/1301.3836.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *J. Artif. Int. Res.* 11(1):1–94.

Brafman, R., and Domshlak, C. 2006. Factored planning: How, when, and when not.

Brafman, R. I.; Shani, G.; and Zilberstein, S. 2013. Qualitative planning under partial observability in multi-agent domains. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, 130–137. AAAI Press.

Brys, T.; Harutyunyan, A.; Vrancx, P.; Taylor, M. E.; Kudenko, D.; and Nowe, A. 2014. Multi-objectivization of reinforcement learning problems by reward shaping. In *2014 International Joint Conference on Neural Networks (IJCNN)*, 2315–2322.

Kurniawati, H.; Hsu, D.; and Lee, W. S. 2008. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *In Proc. Robotics: Science and Systems*.

Laud, A., and DeJong, G. 2003. The influence of reward on the speed of reinforcement learning: An analysis of shaping. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, 440–447. AAAI Press.

Maliah, S.; Brafman, R. I.; and Shani, G. 2017. Increased privacy with reduced communication in multi-agent planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 209–217.

Nair, R.; Tambe, M.; Yokoo, M.; Pynadath, D.; and Marsella, S. 2003. Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. 705–711.

Ng, A. Y.; Harada, D.; and Russell, S. J. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, 278–287. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Nissim, R., and Brafman, R. I. 2014. Distributed heuristic forward search for multi-agent planning. *Journal of Artificial Intelligence Research (JAIR)* 51:293–332.

Oliehoek, F. A., and Amato, C. 2016. *A Concise Introduction to Decentralized POMDPs*. SpringerBriefs in Intelligent Systems. Springer.

Oliehoek, F. A.; Spaan, M. T. J.; Whiteson, S.; and Vlassis, N. 2008. Exploiting locality of interaction in factored dec-pomdps. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '08, 517–524. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Oliehoek, F. A.; Spaan, M. T. J.; Amato, C.; and Whiteson, S. 2013. Incremental clustering and expansion for faster optimal planning in decentralized POMDPs. 46:449–509.

Oliehoek, F. A.; Spaan, M. T. J.; Terwijn, B.; Robbel, P.; and Messias, J. a. V. 2017. The madp toolbox: An open source library for planning and learning in (multi-)agent systems. *J. Mach. Learn. Res.* 18(1):3112–3116.

Oliehoek, F.; Kooij, J.; and Vlassis, N. 2008. The cross-entropy method for policy search in decentralized pomdps. *Informatica (Slovenia)* 32:341–357.

2014. Pomdpx file format (version 1.0).

Sanner, S. 2011. Relational dynamic influence diagram language (rddl): Language description.

Seuken, S., and Zilberstein, S. 2007. Memory-bounded dynamic programming for dec-pomdps. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, 2009–2015. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

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

Shekhar, S.; Brafman, I. R.; and Shani, G. 2019. A factored approach to deterministic contingent multi-agent planning. *ICAPS* 419–427.