Multi-Agent Planning with High-Level Human Guidance[*]

Feng Wu$^{1[0000-0003-3989-0509]}$[**], Shlomo Zilberstein[2], and Nicholas R Jennings[3]

[1]School of Computer Science and Technology, University of Science and Technology of China
[2]College of Information and Computer Sciences, University of Massachusetts Amherst
[3]Department of Computing, Imperial College London
wufeng02@ustc.edu.cn, shlomo@cs.umass.edu,
n.jennings@imperial.ac.uk

Abstract. Planning and coordination of multiple agents in the presence of uncertainty and noisy sensors is extremely hard. A human operator who observes a multi-agent team can provide valuable guidance to the team based on her superior ability to interpret observations and assess the overall situation. We propose an extension of decentralized POMDPs that allows such human guidance to be factored into the planning and execution processes. Human guidance in our framework consists of intuitive high-level commands that the agents must translate into a suitable joint plan that is sensitive to what they know from local observations. The result is a framework that allows multi-agent systems to benefit from the complex strategic thinking of a human supervising them. We evaluate this approach on several common benchmark problems and show that it can lead to dramatic improvement in performance.

Keywords: Multi-Agent Planning, Decentralized POMDP, Human Guidance

1 Introduction

Planning under uncertainty for multi-agent systems is an important and growing area of AI. A common model used to handle such team planning problems is the *Decentralized Partially Observable Markov Decision Process* (DEC-POMDP) [4]. While optimal and approximate algorithms have been developed for DEC-POMDPs [15, 21, 23–25], they assume that the agents' plans remain fixed while they are being executed. Specifically, when a plan computed by these solvers is executed by the agents, it cannot be changed or modified by human operator who may supervise the agents' activities. In real-world problems, this lack of responsiveness and flexibility may increase the likelihood of failure or that the agents damage their workspace or injure people around.

The multi-agent systems community has long been exploring ways to allow agents to get help from humans using various forms of *adjustable autonomy* [5, 8, 10, 14, 19].

[*] This work was supported in part by the National Key R&D Program of China (Grant No. 2017YFB1002204), the National Natural Science Foundation of China (Grant No. U1613216, Grant No. 61603368), and the Guangdong Province Science and Technology Plan (Grant No. 2017B010110011).

[**] Corresponding author.

Human help could come in different forms such as teleoperation [9] or advice in the form of goal bias [6]. Tools to facilitate human supervision of robots have been developed. Examples include a single human operator supervising a team of robots that can operate with different levels of autonomy [3], or robots that operate in hazardous environments under human supervision, requiring teleoperation in difficult situations [11]. However, none of these methods explores these questions with respect to the DEC-POMDP model, with the added challenge that several agents must coordinate based on their partial local information.

In this paper, we focus on a specific setting of DEC-POMDPs in which agents are guided by *runtime high-level commands* from their human operator who supervises the agents' activities. There are several advantages to using high-level commands for guiding agents compared with teleoperation. First, high-level commands are more intuitive and require a lower learning curve for operators. For example, the high-level command "returning home" is much easier for humans to understand and use than the detailed procedure of teleoperating a mobile robot back to its initial location. Second, with high-level commands, operators can focus on the strategic level of thinking while agents take care of the massive low-level sensing and control work (e.g., perception, manipulation, and navigation). By doing so, humans and agents can contribute to the tasks best suited for them. Third, communication between humans and agents usually involves delays and humans need some lead time to respond. Therefore, it is very challenging to teleoperate a system when instant response by the robots to the dynamically changing environment is required, or when there are fewer operators than agents. In contrast, high-level commands such as "searching a building", "cleaning a house", or "pushing a box together", require lower rate of synchronization than teleoperation and can be used to guide the team.

However, planning with high-level human commands for DEC-POMDPs also introduces several challenges. To start, we must allow operators to define useful commands that they will use in the specific domain. The meaning of each command must be encoded in the model so that the solver can interpret it and compute plans for the agents. Furthermore, plans must be represented so that the agents can select actions based on not only their local information but also the command from the operator. In our settings, computing those plans is challenging because we do not know how the operator will command the agents when running the plan. During execution time, similar to teleoperation, we must handle communication delays (although it is less demanding than teleoperation) and help the operators avoid mistakes or unexpected operations.

To this end, we extend the standard finite-horizon DEC-POMDP model and propose HL-DEC-POMDPs, which include humans in the loop of the agents' decision making process. More specifically, we provide a new model that allows operators to define a set of high-level commands. Each command has a specific context that can be easily understood by human operators. These commands are designed for situations where the operator can provide useful guidance. We present planning algorithms for this new model to compute plans conditioned on both the local information of an agent and the command initiated by the operator. In the execution phase, the operator interacts with the agents with the high-level command similar to teleoperation: the operator observing the agents' activities initiates commands and the agents follow a plan based on the command from the operator. In fact, teleoperation can be viewed as a special case of our approach where each low-level control operation is mapping to a high-level command. We also provide a mechanism for handling delays and an algorithm to suggest feasible commands to the operator. This is helpful for the operator to select the best command and avoid mistakes. This is the first work to bring humans in the loop of multi-agent planning under the framework of DEC-POMDPs. We contribute a novel model to consider human supervision of autonomous agents and an efficient algorithm to compute human-in-the-loop plans.

2 Related Work

In terms of guiding agents with high-level commands, our work is similar to the coaching system in RoboCup soccer simulation where a coach who gets an overview of the whole game sends commands to the players of its own team. However, the coach is also a software agent (there is no human in the loop) and its decision is mainly on recognizing and selecting the opponent model [12, 17]. For planning with human guidance, MAPGEN [1], the planning system for Mars rover missions, allows operators to define constraints and rules for a plan, which are subsequently enforced by automated planners to produce the plan. However, this approach is only for single-agent problems and does not consider the uncertainty in the environment. It is not clear how this can be done for DEC-POMDPs.

For human-robot interaction, there has also been research on mobile robots that can proactively seek help from people in their environment to overcome their limitations [18,26]. Researchers have started to develop robots that can autonomously identify situations in which a human operator must perform a subtask [22] and design suitable interaction mechanisms for the collaboration [26].

3 The HL-DEC-POMDP Model

Our model is an extension of the standard DEC-POMDP. Before presenting our model, we first briefly review the DEC-POMDP model. Formally, a *Decentralized Partially Observable Markov Decision Process* (DEC-POMDP) is defined as a tuple $\langle I, S, \{A_i\}, \{\Omega_i\}, P, O, R\rangle$, where:

- I is a set of n agents where $|I| = n$ and each agent has a unique ID number $i \in I$.
- S is a set of states and $b^0 \in \Delta(S)$ is the initial state distribution where $b^0(s)$ is the probability of $s \in S$.
- A_i is a set of actions for agent i. Here, we denote $\boldsymbol{a} = \langle a_1, a_2, \cdots, a_n\rangle$ a joint action where $a_i \in A_i$ and $\boldsymbol{A} = \times_{i\in I} A_i$ the set of joint actions where $\boldsymbol{a} \in \boldsymbol{A}$.
- Ω_i is a set of observations for agent i. Similarly, we denote $\boldsymbol{o} = \langle o_1, o_2, \cdots, o_n\rangle$ a joint observation where $o_i \in \Omega_i$ and $\boldsymbol{\Omega} = \times_{i\in I} \Omega_i$ the joint set where $\boldsymbol{o} \in \boldsymbol{\Omega}$.
- $P : S\times \boldsymbol{A}\times S \to [0, 1]$ is the Markovian transition function and $P(s'|s, \boldsymbol{a})$ denotes the probability distribution of the next state s' when agents take $\boldsymbol{a}$ in s.
- $O : S \times \boldsymbol{A} \times \boldsymbol{\Omega} \to [0, 1]$ is the observation function and $O(\boldsymbol{o}|s', \boldsymbol{a})$ denotes the probability distribution of observing $\boldsymbol{o}$ after taking $\boldsymbol{a}$ with outcome state s'.

– $R : S \times \boldsymbol{A} \to \Re$ is the reward function and $R(s, \boldsymbol{a})$ is the immediate reward of the team when all the agents take joint action $\boldsymbol{a}$ in state s.

Now, we turn to our model. A *Decentralized Partially Observable Markov Decision Process with Humans in the Loop* (HL-DEC-POMDP) is defined as a tuple $\mathcal{M} = \langle I, S, C, \{A_i\}, \{\Omega_i\}, P, O, R \rangle$, with the following additional component C and modification of the reward function R:

– C is a set of high-level commands for human operators. We assume that when a command $c \in C$ is initiated by the operator it can be received by all the agents.
– $R : S \times C \times \boldsymbol{A} \to \Re$ is the reward function and $R(s, c, \boldsymbol{a})$ is the immediate reward of the team when all the agents take $\boldsymbol{a}$ in state s with command c.

In the execution phase of HL-DEC-POMDPs, the operator can initiate a command $c \in C$ to the agents. Thus, each agent i can make its decision based on both its local observation o_i from the environment and the command c from the operator. Intuitively, HL-DEC-POMDP is at least as hard as the standard DEC-POMDP (i.e., NEXP-hard) since DEC-POMDP is a special case of our model with only one command (i.e., $|C| = 1$). In this model, the operator can observe the agents' activities and guide them with predefined commands $c \in C$.

To intuitively explain what the high-level commands are and how they work in HL-DEC-POMDPs, we use the cooperative box-pushing problem [20] as example. This is a common DEC-POMDP benchmark problem where two agents in a grid world must coordinate to independently push small boxes or cooperatively push the large box. In this problem, a possible set of commands for the operator could be $C = \{$*"pushing a large box", "pushing small boxes", "automatic"*$\}$. Each command has a specific meaning, which is intuitive for people familiar with the problem. Moreover, they all focus on the high-level decisions that can be used straightforwardly by the operator to guide the agents. The design of commands depends on the requirements of the operator on her supervision tasks. For example, if the operator wants the agents to push the two small boxes separately, the command *"pushing small boxes"* can be split into two commands as *"pushing the left small box"* and *"pushing the right small box"*.

In our model, the meaning of each command $c \in C$ is specified in the reward function R. Given a command, we can generate a reward function to achieve the desirable behavior of the agents similar to building the reward model for standard DEC-POMDPs. For example, when $c =$*"pushing small boxes"*, R with c is defined so that only pushing small boxes has positive rewards. Planning with this reward function will output a plan that lets all the agents go for the small boxes. Similarly, if $c =$*"pushing a large box"*, only pushing a large box is rewarded in R. If the operator sets the command $c =$*"automatic"*, the agents will push boxes as the original box-pushing problem (i.e., the large box has higher reward than the small boxes). In this example, each command is defined for all the agents. Indeed, the command set can be augmented to include more complex commands so that each agent gets a specific instruction. For example, a command can be $c =$*"agent A pushing a small box and agent B, C pushing the large box"*. Similarly, the reward function R can be specified for this command.

Notice that the command initiated by operators does not affect the transition model of the states but only the reward received by the agents (i.e., the reward function R in the model). By so doing, we assume that the operator cannot directly interact with the environment using the commands. Instead, the operator can guide the agents with the commands to achieve expected behaviors of the agents. Typical scenarios of our setting include operators situated in a base station remotely supervising the agents in some workspace. It is worth pointing out that each command can be efficiently transferred to the agents with its index because C is predefined and known for all the agents.

In HL-DEC-POMDP, a local policy q_i for agent i is a conditional rule mapping from its observation-action history $h_i \in H_i$ and the command $c \in C$ to an action $a_i \in A_i$, i.e., $q_i : H_i \times C \to A_i$. A joint policy $\boldsymbol{q} = \langle q_1, q_2, \cdots, q_n \rangle$ is a collection of local policies, one for each agent. The goal of solving a HL-DEC-POMDP is to find a joint policy $\boldsymbol{q}^*$ for the agents that maximizes the expected value:

$$V(b^0, c^0, \boldsymbol{q}^*) = \mathbb{E}\left[\sum_{t=0}^{T-1} R^t \middle| b^0, c^0, \boldsymbol{q}^*\right] \tag{1}$$

where c^0 is the initial command initiated by the operator.

4 Solving HL-DEC-POMDPs

In the HL-DEC-POMDP model, we represent our policies by stochastic policy trees where the nodes and branches are parameterized by probability distributions. Specifically, a stochastic policy for agent i is defined recursively as: $q_i = \langle \pi_i, \lambda_i \rangle$, where:

- π_i is an action selection function that specifies a distribution over the actions. $\pi_i(a_i|q_i, c)$ denotes the probability of selecting action a_i in node $q_i \in Q_i$ with command c.
- λ_i is a node transition function that defines a distribution over the sub-trees. $\lambda_i(q'_i|q_i, o_i, c)$ denotes the probability of selecting node q'_i (a sub-tree with q'_i as its root) when o_i is observed and command c is given by the operator.

Note that the action selection function is defined at the root node of q_i and the node transition function is defined for the branches of the root node where q'_i is a sub-policy of q_i after observing o_i with c. Here, c is the command for selecting the policy q'_i. Indeed, a (deterministic) policy tree is a special case of our stochastic policy. At each node q_i, the agent executes an action a_i sampled from $\pi_i(\cdot|q_i, c)$. Based on its observation o_i and the command c, it will transition to a new node q'_i sampled from the function $\lambda_i(\cdot|q_i, o_i, c)$.

Given a state s and a command c, the expected value of a joint policy $\boldsymbol{q}$ represented by our stochastic policy trees can be computed recursively by the Bellman equation:

$$\begin{aligned} V^t(s, c, \boldsymbol{q}) = \sum_{\boldsymbol{a} \in \boldsymbol{A}} \prod_{i \in I} \pi_i(a_i|q_i, c) \Bigg[R(s, c, \boldsymbol{a}) + \sum_{s' \in S} P(s'|s, \boldsymbol{a}) \sum_{\boldsymbol{o} \in \boldsymbol{\Omega}} O(\boldsymbol{o}|s', \boldsymbol{a}) \\ \sum_{\boldsymbol{q}'} \prod_{i \in I} \lambda_i(q'_i|q_i, o_i, c) \sum_{c' \in C} \delta^t(c'|\cdot) \cdot V^{t+1}(s', c', \boldsymbol{q}') \Bigg] \end{aligned} \tag{2}$$

where $\delta^t(c'|\cdot)$ is the distribution of choosing command c'.

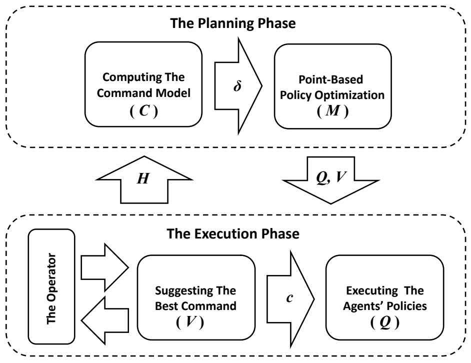
The Planning Phase
Computing The Command Model (C)
δ
Point-Based Policy Optimization (M)
H
Q, V
The Execution Phase
The Operator
Suggesting The Best Command (V)
c
Executing The Agents' Policies (Q)

Fig. 1. Overview of our Framework

The basic framework of our approach is shown in Figure 1. In the planning phase, we first generate a command model δ^t and then compute the policies and values. In the execution phase, we take the command input of the operator and compare it with the command computed by our algorithm. If they are different, we suggest our command to the operator and ask her to confirm or amend her choice. Next, her choice is sent to the agents who will execute the policies based on the command from the operator. The following sections will given more detail on our algorithms.

4.1 The Command Model

As shown in Equation 2, the command model is a rule of selecting commands. However, commands in our approach are actually selected by the operator at runtime. Thus, a model of how command decisions are made by the operator is needed for the planning phase so that our planner can optimize the agents' policies. Although human decisions may depend on many complex factors (not only their perspective of the problem but also their expertise and experience), we assume that the next command in our model is selected only based on the current command and the next state at a single point in time. Other inputs such as the history of the previous commands and indications of the operator's attentiveness are not used. Although a richer representation might improve the predictive quality, it will dramatically increase the computational complexity of learning and planning. Therefore, we leave an investigation of the correct balance of representational richness and simplicity for future work. Specifically, we define the command model as $\delta^t : C \times S \times C \rightarrow [0, 1]$ where $\delta^t(c'|c, s')$ is a probability of selecting command c' for the next step given the current command c and the next state s'. Here, we use probability distributions to model human decisions because whether a command is selected and its likelihood appears to be highly stochastic in our problems. Note that the command model is only used in the planning phase. The operator does not need to know the state to select her commands during execution time. There are several methods to specify δ^t, depending on the characteristic of the problem domain and the role of the operator.

A Fixed Command Model In a *fixed command model* the command is assumed that any command issued by an operator will remain the same for the rest of the decision steps, resulting in the following simple command model:

$$\delta^t(c'|c,s') = \begin{cases} 1 \; c' = c \\ 0 \; c' \neq c \end{cases} \tag{3}$$

The policies computed with this command model allow the operator to switch among different reward models (i.e., objectives) during execution time. Once a command is selected by the operator, the agents will stick to that "mode" until a different command is issued. For example, in the cooperative box-pushing problem, if a command is set for the agents to push small boxes, they will repeatedly push small boxes until they are allowed to push the large box. If the policies for each reward model are independently computed, it is nontrivial for the agents to switch to other policies in the execution phase given the partial observability of the agents in DEC-POMDPs. Therefore, our approach is more sophisticated given that our policies straightforwardly allow the agents to transition to other "mode" without re-coordination.

A Learned Command Model Another option is to learn the command model from a log of data collected in previous trial executions. The data log records the joint action-observation history of the agents, the obtained rewards, and the commands initiated by the operators: H=$(c^0, \boldsymbol{a}^0, r^0, \boldsymbol{o}^1, c^1, \boldsymbol{a}^1, r^1, \boldsymbol{o}^2, \cdots, c^{T-1}, \boldsymbol{a}^{T-1}, r^{T-1})$. Given this, the operators do offline analysis of the data and evaluate the agents' performance. In this process, additional rewards could be specified by the operators, which may include some of the operators' evaluation on the agents' behaviors that is not captured by the model. For example, if a robot injured people in the environment when doing a task, a penalty should be given to it by the operators. At the end of the analysis, the rewards in H are replaced by mixtures of the original rewards and the rewards specified by the operators.

Given the data evaluated by the operators, we can learn a new command model δ^t that maximizes the expected value. Because the model and joint policies $\boldsymbol{q}$ are known for the previous executions, the parameters of δ^t can be optimized by a gradient ascent method similar to [16] using the agents' history data. Specifically, Equation 1 can be rewritten with the histories in H as follow:

$$V(b^0, c^0, \boldsymbol{q}) = \sum_{t=0}^{T-1} \sum_{h^t \in H^t} Pr(h^t | b^0, c^0, \boldsymbol{q}) r^t(h^t) \tag{4}$$

where $H^t \subseteq H$ is the histories up to time t, $r^t(h^t)$ is the reward given by the end of h^t, and $Pr(h^t|b^0, c^0, \boldsymbol{q})$ is the probability for h^t that can be computed given the model and policies. Then, we can calculate the derivative of V for each δ^t and do a gradient ascent on V by making updates $\Delta\delta^t = \beta \partial V(b^0, c^0, \boldsymbol{q}) / \partial \delta^t$ with step size β.

4.2 Point-Based Policy Optimization

As aforementioned, our HL-DEC-POMDP model is as hard as the DEC-POMDP (i.e., NEXP-hard). Therefore, optimal algorithms are mostly of theoretical significance. To

Algorithm 1: Point-Based DP for HL-DEC-POMDPs

```
Input: the HL-DEC-POMDP model $\mathcal{M}$.
Output: the best joint policy $\boldsymbol{Q}^0$.
for $t = T - 1$ to 0 do
    $\boldsymbol{Q}^t \leftarrow \emptyset$
    for $k = 1$ to $N$ do
        $(b, d) \leftarrow$ sample a joint belief state $b$ and a command distribution $d$ up to the
          current step $t$
        // Equation 5
        $\boldsymbol{q} \leftarrow$ compute the best policy with $(b, d)$
        $\boldsymbol{Q}^t \leftarrow \boldsymbol{Q}^t \cup \{\boldsymbol{q}\}$
    // Equation 6
    $V^t \leftarrow$ evaluate $\boldsymbol{Q}^t$ for $\forall s \in S, c \in C, \boldsymbol{q} \in \boldsymbol{Q}^t$
return $\boldsymbol{Q}^0$
```

date, state-of-the-art optimal approaches can only solve DEC-POMDP benchmark problems with very short horizons [7, 15]. To solve large problems, one of the popular techniques in the DEC-POMDP literature is using Memory-Bounded DP (MBDP) [21] — a variation of the DP algorithm. At each iteration, it first backups the policies of the previous iteration as the standard DP. Then it generates a set of belief points and only retains the polices that have the highest value on those points for the next iteration. By doing so, the number of possible policies at each iteration does not blow up with the horizon. Several successors of MBDP have improved significantly the performance of the approach, particularly the point-based DP technique [13], which we build on to compute policies for our extended problem representation.

The main process is outlined in Algorithm 1. Similar to DP, the policy is optimized backwards from the last step to the first one. At each iteration (Lines 1-8), we first sample N pairs of (b, d) from the first step down to the current step (Line 5) where $b \in \Delta(S)$ is a probability distribution over the state space S and $d \in \Delta(C)$ is a probability distribution over the command set C. Then, we compute a joint policy for each sampled (b, d) pair (Line 6). Sampling can be performed efficiently by running simulations on heuristic policies. The heuristic policy can be either the policy obtained by solving the underlying MDP or just a random policy where agent i's action is uniformly selected from its action set A_i. In the underlying MDP, the command is treated as a state variable. Hence the state space of the underlying MDP is $\mathbb{S} = S \times C$ and its transition function is $\mathbb{P}(s', c'|s, c, \boldsymbol{a}) = \delta^t(c'|c, s')P(s'|s, \boldsymbol{a})$. This is a standard MDP that can be solved by dynamic programming. A simple technique to improve sampling efficiency is to use a *portfolio* of different heuristics [21].

In each simulation of the t-th DP iteration, we first select an initial command c and draw a state $s \sim b^0(\cdot)$ from the initial state distribution. Next, we compute a joint action $\boldsymbol{a}$ using the heuristic. Then, we sample the next state $s' \sim P(\cdot|s, \boldsymbol{a})$ based on the transition function and draw the next command $c' \sim \delta^t(\cdot|c, s')$ from the command model. This process continues with the state $s \leftarrow s'$ and the command $c \leftarrow c'$ until the sampling horizon is reached. Here, for the t-th DP iteration, the sampling horizon is

(T-t-1). In the last step of the simulation, the state s and command c are recorded with $b(s) \leftarrow b(s) + 1$ and $d(c) \leftarrow d(c) + 1$. We repeat the simulation K times and produce the distributions (b, d) by averaging the samples: $b(s) \leftarrow b(s)/K$ and $d(c) \leftarrow d(c)/K$. According to the central limit theorem, the averaged values (b, d) will converge to the true distributions of states and commands as long as K is sufficiently large.

Give each sampled pair (b, d), the best joint policy $\boldsymbol{q}$ can be computed by solving the following optimization problem:

$$\begin{aligned}
\max_{\pi_i,\lambda_i} & \textstyle\sum_{s\in S} b(s) \sum_{c\in C} d(c) \sum_{\boldsymbol{a}\in \boldsymbol{A}} \prod_{i\in I} \pi_i(a_i|q_i, c)\Big[R(s, c, \boldsymbol{a}) + \sum_{s'\in S} P(s'|s, \boldsymbol{a}) \\
& \textstyle\sum_{\boldsymbol{o}\in\boldsymbol{\Omega}} O(\boldsymbol{o}|s', \boldsymbol{a}) \sum_{\boldsymbol{q}'} \prod_{i\in I} \lambda_i(q'_i|q_i, o_i, c) \sum_{c'\in C} \delta^t(c'|c, s')V^{t+1}(s', c', \boldsymbol{q}')\Big] \\
\text{s.t.} \quad & \forall i, c, a_i,\ \ \pi_i(a_i|q_i, c) \geq 0, \forall i, c, \textstyle\sum_{a_i\in A_i} \pi_i(a_i|q_i, c) = 1 \\
& \forall i, c, o_i, q'_i,\ \ \lambda_i(q'_i|q_i, o_i, c) \geq 0, \forall i, c, o_i, \textstyle\sum_{q'_i\in Q'_i} \lambda_i(q'_i|q_i, o_i, c) = 1
\end{aligned} \tag{5}$$

where the variables π_i and λ_i are the parameters of agent i's policy and the objective is to maximizes the expected value $V^t(b, d, \boldsymbol{q}) = \sum_{s\in S} b(s) \sum_{c\in C} d(c)V^t(s, c, \boldsymbol{q})$. The constraints in Equation 5 guarantee that the optimized policy parameters π_i and λ_i are probability distributions (i.e., all distributions are non-negative and sum to 1).

After a set of policies are generated, we evaluate each joint policy $\boldsymbol{q} \in \boldsymbol{Q}^t$. Specifically, we compute the expected values as defined in Equation 2 for every state $s \in S$ and command $c \in C$ (Line 8) as follow:

$$\begin{aligned}
V^t(s, c, \boldsymbol{q}) = \sum_{\boldsymbol{a}\in\boldsymbol{A}} \prod_{i\in I} \pi_i(a_i|q_i, c) \Bigg[& R(s, c, \boldsymbol{a}) + \sum_{s'\in S} P(s'|s, \boldsymbol{a}) \sum_{\boldsymbol{o}\in\boldsymbol{\Omega}} O(\boldsymbol{o}|s', \boldsymbol{a}) \\
& \sum_{\boldsymbol{q}'} \prod_{i\in I} \lambda_i(q'_i|q_i, o_i, c) \sum_{c'\in C} \delta^t(c'|c, s')V^{t+1}(s', c', \boldsymbol{q}')\Bigg]
\end{aligned} \tag{6}$$

where V^{t+1} is the value obtained in the previous iteration.

All the above computations assume that the operator can make decisions and initiate commands at the same rate as the agents (i.e., at each step, the operator initiates a command and then each agent selects an action). This might be unrealistic in practice for two reasons: First, the decision cycle of autonomous agents could be much faster. For example, a mobile robot can move very quickly and response to the environment in a fraction of a second. However, depending on the problem, the operator usually takes a few seconds or even minutes to response (she may need to scan the environment and understand the current situation and then issues a command by pressing a button). Second, the communication between operators and agents could introduce a delay. When a operator is situated at a distant base station, it may take some time for the state information to be transferred and displayed on the operator's screen. Similarly, it may also take time for the operator's command to be transferred to the agents. Depending on the distance between the operator and agents, the communication delay may range from a few seconds to several hours (e.g., when communicating with space exploration rovers). Thus, it is more realistic to assume that the operator's decisions are made at a lower rate.

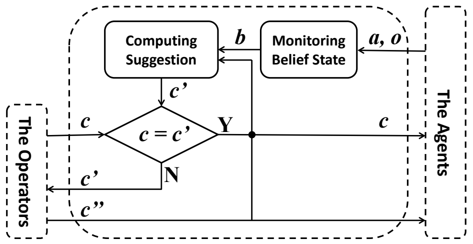

Fig. 2. The Command Suggestion System

Specifically, we assume that the operator's decision is made at every interval of τ steps (i.e., $0, \tau, 2\tau, 3\tau, \cdots$) up to the horizon T. Within an interval, the command is initiated by the operator at the beginning and remains fixed until the end. To compute the policies, our sampling algorithm must be adapted to only draw a new command $c' \sim \delta^t(\cdot|c, s')$ at the beginning of an interval. Similarly, in Equations 5 and 6, if the time step t is at the beginning of an interval, we use the same command model to optimize and evaluate the policies. Otherwise, we use the fixed command model defined by Equation 3 to keep the command fixed within the interval. Indeed, if $\tau = T$, the fixed model is actually used for the whole planning process.

4.3 Suggesting Commands to the Operators

In the execution phase, the operator guides the agents by selecting commands. We assume she takes the full responsibility for every command initiated by her. Nevertheless, it will be useful if our system can verify her choice and give her suggestions when necessary. The operator may make mistakes, especially when she becomes distracted or tired after long shift. She may also neglect some key factors of the current situation that may affect her decision (e.g., low battery level of some robots). Therefore, it will be helpful if our system can remind her or provide an alternative choice that might be better than the operator's original command. Given the suggestion computed by our system, the operator can evaluate the suggested command and her original command and make the final decision.

Figure 2 illustrates our suggestion mechanism for the operator. As we can see, the operator first makes her decision and initiate command c. Then, our system computes a command c' based on its current information about the agents b. If this command c' is different from the operator's choice c, our system will present its suggestion c' to the operator and ask her to confirm or amend her choice. The operator can insist on her original command if she feels confident about it or selects another command. Her final decision c'' is sent to the agents. We deliberately postpone our suggestions to the operator so that her decisions are not biased by the suggestions. Our system does not intend to replace the operator and make decisions for her. Instead, it is designed to provide a chance for her to correct mistakes (if any) or improve her decision (if possible). Notice that the suggestions are computed by a software agent running on the operator's system (e.g., computers at her base station) where an overview of the whole agent team is available as for the operator.

To give suggestions, we compute in the planning phase the values of selecting every command $c' \in C$ as follows:

$$V^t(s, c, \boldsymbol{q}; c') = \sum_{\boldsymbol{a} \in \boldsymbol{A}} \prod_{i \in I} \pi_i(a_i|q_i, c) \Bigg[R(s, c, \boldsymbol{a}) + \sum_{s' \in S} P(s'|s, \boldsymbol{a}) \sum_{\boldsymbol{o} \in \boldsymbol{\Omega}} O(\boldsymbol{o}|s', \boldsymbol{a}) \sum_{\boldsymbol{q}'} \prod_{i \in I} \lambda_i(q'_i|q_i, o_i, c) V^{t+1}(s', c', \boldsymbol{q}') \Bigg] \tag{7}$$

where V^{t+1} is the expected value computed by Equation 6 in Algorithm 1. Then, in the execution phase, our system computes the best next command c' (from the system's perspective) that maximizes this value as:

$$c' = \arg\max_{c \in C} \sum_{s \in S} b^t(s) V^t(s, c^t, \boldsymbol{q}; c) \tag{8}$$

This command c' will be compared with the operator's original choice c. If they are different, c' will be suggested to the operator. We can also present to the operator the difference in value between these two commands for reference:

$$D(c||c') = \sum_{s \in S} b^t(s)[V^t(s, c^t, \boldsymbol{q}; c') - V^t(s, c^t, \boldsymbol{q}; c)] \tag{9}$$

Notice that both Equations 8 and 9 require knowledge of the current joint belief state $b^t(s)$. As shown in Figure 2, this joint belief state is monitored and updated in our suggestion system during execution time. Here, the joint belief state is computed recursively by the Bayesian rule:

$$b^{t+1}(s') = \alpha \, O(\boldsymbol{o}^{t+1}|s', \boldsymbol{a}^t) \sum_{s \in S} b^t(s) P(s'|s, \boldsymbol{a}^t) \tag{10}$$

where b^t is the previous belief state (b^0 is the initial state distribution), $\boldsymbol{o}^{t+1}$ is the agents' latest joint observation, $\boldsymbol{a}^t$ is a joint action taken by the agents at the previous step, and $\alpha = 1/\sum_{s' \in S} b^{t+1}(s')$ is the normalization factor. Once the up-to-date information is transferred back from the agents to the operator, the joint belief state can be updated by Equation 10. Indeed, our suggestion system is a POMDP agent that takes the same input as the operator (i.e., the agents' $\boldsymbol{a}$ and $\boldsymbol{o}$), maintains a belief b^t over the current state, and makes suggestions to the operator based on the expected value V^t of the agents' policies.

5 Experiments

We implemented our algorithm and tested it on two common benchmark problems previously used for DEC-POMDPs: Meeting in a 3×3 Grid [2] and Cooperative Box-Pushing [20]. For each problem, we first designed a set of high-level commands C and

Table 1. Results for two Benchmark Problems

τ	**LEARNED**	**FIXED**	**MBDP**	**LEARNED**	**FIXED**	**MBDP**
	Meeting in a 3×3 Grid (T=20)			Cooperative Box-Pushing (T=20)		
1	189.69, 1.87	133.70, 1.32	24.95, 0.23	45.10, 0.7%	43.48, 0.9%	-11.34, 87.1%
3	175.28, 1.73	114.88, 1.13	-	26.31, 4.2%	18.17, 4.9%	-
7	154.34, 1.51	94.12, 0.92	-	11.82, 35.5%	9.75, 42.7%	-

the corresponding reward function R as in our HL-DEC-POMDP model. Then, we ran our planning algorithm to compute policies for the agents. During execution time, we let a person (the operator) guide the agents with the commands in C while the agents execute the computed policies accordingly. The suggestions for the operator are computed during the process.

We invited 30 people to participate in our tests as operators. Before the tests, they were given tutorials on the domains, what they should do for each domain, and how they can command the simulated robots using our interface. Then, we divided them into two groups. The first 5 people were asked to guide the agents given the policies computed using the fixed command model. We recorded their operations and learned a new command model from the logged data. After that, we asked the second group to command the agents with the policies computed using the learned model. Each person guided the agents with different intervals (τ=1, 3, 7) and repeated each individual test 10 times. The averaged performance of the two groups was reported for each test. To show how agents benefit from high-level human guidance, we also present the results of the flat MBDP policies on our domains where no human decisions are involved.

The results were summarized in Table 1. For the first problem, the first value in a cell is the overall reward and the second value shows how many times on average the robots met at the highly rewarded corners in a test. For the second problem, the first value is also the overall reward while the second value shows the percentage of the total tests when the animal got injured by the robots. From the table, we can see that in both domains human guidance can produce dramatic improvements in performance over the flat MBDP policies and the learned command model does produce significant additional performance gains. For example, in the second domain, the animals were very likely to be injured without human guidance (87.1%) while the chance with our high-level commands reduced to less than 1%. Note that the robots' actions in our first domain are very stochastic (with only the success rate of 0.6). Our values are also significantly better than the MBDP policies in this domain. Without human guidance, we observed that the robots met equally at one of the 4 corners. Overall, our results confirmed the advantage of human guidance in agents' plans.

Additionally, communication delays (τ) were well handled by our high-level commands. With τ=7, the operators only allowed to command the agents 2 times within the total 20 steps. However, our values with guidance are still significantly better than the results without guidance. In contrast, we observed that tele-operation with the same delays made no difference to the results due to the problem uncertainty. In the tests, we observed that our suggested mechanisms were useful to the operators especially when the problem was reset (once agents meet at a corner in the first problem or a box is pushed to the goal location in the second). Most of the operators were not aware of the change until they were asked to confirm their commands. We also observed that the commands were more frequently modified after the participants repeated their test 6 to 7 times and they did become more likely to make mistakes when felt tired.

6 Conclusions

We introduce the HL-DEC-POMDP model — a novel extension of DEC-POMDPs to incorporate high-level human guidance in the agents' plans. Specifically, our model allows the operators of the agents to define a set of high-level commands that are intuitive to them and useful for their daily supervision. We also presented algorithms that can compute the plans for the operators to guide the agents with those commands during execution time. This enables the agents to take advantage of the operators' superior situation awareness. This is nontrivial because the agents' policies do not only depend on the operators' commands but also on their local information and how this will affect the decision of the other agents. Moreover, our model is more robust to communication delays than simple teleoperation because commands need only provide high-level guidance. In our planning algorithms, the quality of agents' plans can be improved by learning from the operators' experience. In our experiments, whenever the operators have information or knowledge that is not captured in the agents' plans, significant improvements in agents' performance have been observed with high-level human guidance. In the future, we plan to test our model and algorithm on larger domains where high-level human guidance could play a crucial role.

References

1. Ai-Chang, M., Bresina, J., Charest, L., Chase, A., Hsu, J.J., Jonsson, A., Kanefsky, B., Morris, P., Rajan, K., Yglesias, J., Chafin, B.G., Dias, W.C., Maldague, P.F.: MAPGEN: Mixed-initiative planning and scheduling for the mars exploration rover mission. IEEE Intelligent Systems **19**(1), 8–12 (2004)
2. Amato, C., Dibangoye, J.S., Zilberstein, S.: Incremental policy generation for finite-horizon DEC-POMDPs. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling. pp. 2–9 (2009)
3. Bechar, A., Edan, Y.: Human-robot collaboration for improved target recognition of agricultural robots. Industrial Robot: An International Journal **30**(5), 432–436 (2003)
4. Bernstein, D.S., Givan, R., Immerman, N., Zilberstein, S.: The complexity of decentralized control of Markov decision processes. Mathematics of Operations Research **27**(4), 819–840 (2002)
5. Bradshaw, J.M., Jung, H., Kulkarni, S., Johnson, M., Feltovich, P., Allen, J., Bunch, L., Chambers, N., Galescu, L., Jeffers, R., et al.: Kaa: Policy-based explorations of a richer model for adjustable autonomy. In: Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems. pp. 214–221 (2005)
6. Côté, N., Canu, A., Bouzid, M., Mouaddib, A.I.: Humans-robots sliding collaboration control in complex environments with adjustable autonomy. In: Proceedings of Intelligent Agent Technology (2013)

7. Dibangoye, J.S., Amato, C., Buffet, O., Charpillet, F.: Optimally solving Dec-POMDPs as continuous-state MDPs. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (2013)
8. Dorais, G., Bonasso, R.P., Kortenkamp, D., Pell, B., Schreckenghost, D.: Adjustable autonomy for human-centered autonomous systems. In: IJCAI Workshop on Adjustable Autonomy Systems. pp. 16–35 (1999)
9. Goldberg, K., Chen, B., Solomon, R., Bui, S., Farzin, B., Heitler, J., Poon, D., Smith, G.: Collaborative teleoperation via the internet. In: Proceedings of the 2000 IEEE International Conference on Robotics and Automation. vol. 2, pp. 2019–2024 (2000)
10. Goodrich, M.A., Olsen, D.R., Crandall, J.W., Palmer, T.J.: Experiments in adjustable autonomy. In: Proceedings of IJCAI Workshop on Autonomy, Delegation and Control: Interacting with Intelligent Agents. pp. 1624–1629 (2001)
11. Ishikawa, N., Suzuki, K.: Development of a human and robot collaborative system for inspecting patrol of nuclear power plants. In: Proceedings 6th IEEE International Workshop on Robot and Human Communication. pp. 118–123 (1997)
12. Kuhlmann, G., Knox, W.B., Stone, P.: Know thine enemy: A champion robocup coach agent. In: Proceedings of the 21st National Conference on Artificial Intelligence. pp. 1463–1468 (2006)
13. Kumar, A., Zilberstein, S.: Point-based backup for decentralized POMDPs: Complexity and new algorithms. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. pp. 1315–1322 (2010)
14. Mouaddib, A.I., Zilberstein, S., Beynier, A., Jeanpierre, L.: A decision-theoretic approach to cooperative control and adjustable autonomy. In: Proceedings of the 19th European Conference on Artificial Intelligence. pp. 971–972 (2010)
15. Oliehoek, F.A., Spaan, M.T., Amato, C., Whiteson, S.: Incremental clustering and expansion for faster optimal planning in decentralized pomdps. Journal of Artificial Intelligence Research **46**, 449–509 (2013)
16. Peshkin, L., Kim, K.E., Meuleau, N., Kaelbling, L.P.: Learning to cooperate via policy search. In: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence. pp. 489–496 (2000)
17. Riley, P.F., Veloso, M.M.: Coach planning with opponent models for distributed execution. Autonomous Agents and Multi-Agent Systems **13**(3), 293–325 (2006)
18. Rosenthal, S., Veloso, M.M.: Mobile robot planning to seek help with spatially-situated tasks. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence (2012)
19. Scerri, P., Pynadath, D., Tambe, M.: Adjustable autonomy in real-world multi-agent environments. In: Proceedings of the 5th International Conference on Autonomous agents. pp. 300–307 (2001)
20. Seuken, S., Zilberstein, S.: Improved memory-bounded dynamic programming for decentralized POMDPs. In: Proceedings of the 23rd Conference Conference on Uncertainty in Artificial Intelligence. pp. 344–351 (2007)
21. Seuken, S., Zilberstein, S.: Memory-bounded dynamic programming for DEC-POMDPs. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence. pp. 2009–2015 (2007)
22. Shiomi, M., Sakamoto, D., Kanda, T., Ishi, C.T., Ishiguro, H., Hagita, N.: A semi-autonomous communication robot: a field trial at a train station. In: Proceedings of the 3rd ACM/IEEE International Conference on Human Robot Interaction. pp. 303–310. ACM, New York, NY, USA (2008)
23. Szer, D., Charpillet, F.: Point-based dynamic programming for DEC-POMDPs. In: Proceedings of the 21st National Conference on Artificial Intelligence. pp. 1233–1238 (2006)

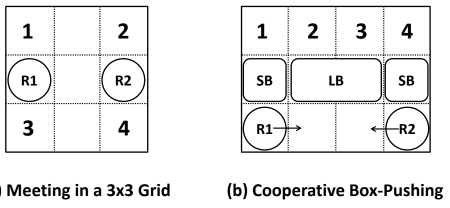

Fig. 3. The Benchmark Problems

24. Wu, F., Jennings, N.R., Chen, X.: Sample-based policy iteration for constrained dec-pomdps. In: Proceedings of the 20th European Conference on Artificial Intelligence (ECAI). pp. 858–863 (2012)
25. Wu, F., Zilberstein, S., Chen, X.: Trial-based dynamic programming for multi-agent planning. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence. pp. 908–914 (2010)
26. Yanco, H.A., Drury, J.L., Scholtz, J.: Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition. Human–Computer Interaction **19**(1-2), 117–149 (2004)

A The Benchmark Problems

A.1 Meeting in a 3×3 Grid

In this problem, as shown in Figure 3(a), two robots R1 and R2 situated in a 3×3 grid try to stay in the same cell together as fast as possible. There are 81 states in total since each robot can be in any of the 9 cells. They can move *up*, *down*, *left*, *right*, or *stay* so each robot has 5 actions. Their moving actions (i.e., the actions except *stay*) are stochastic. With probability 0.6, they can move in the desired direction. With probability 0.1, they may move in another direction or just stay in the same cell. There are 9 observations per robot. Each robot can observe if it is near one of the corners or walls. The robots may meet at any of the 4 corners. Once they meet there, a reward of 1 is received by the agents. To make the problem more challenging, the agents are reset to their initial locations when they meet at the corners.

We design the high-level commands so that the robots are asked to meet at a specific corner. In more detail, the command set for this problem is C=$\{c_0, c_1, c_2, c_3, c_4\}$ where c_0 allows the agents to meet at any corner and $c_i (i \neq 0)$ is the command for the robots to meeting in the corner labeled with i in Figure 3(a). Depending on what we want to achieve in the problem, the commands can be more general (e.g., meeting at any of the top corners) or specific (e.g., meeting at the top-left corner without going through the center). The reward function is implemented so that they are rewarded only when they meet at the specific corner (for c_0, the original reward function is used). For example, $R(c_1, \cdot, \cdot) = 1$ only when they meet at the top-left corner and 0 otherwise.

During execution time, we simulate a random event to determine whether meeting at one of the corners has a much higher reward (i.e., 100) and which corner. It is a *finite state machine* (FSM) with 5 states where state 0 means there is no highly rewarded corner and state i ($1 \le i \le 4$) means the corner labeled with i has the highest reward. The transition function of this FSM is predetermined and fixed for all the tests, but it is not captured by the model and not known during planning time. Therefore, they are not considered in the agents' policies. The event can only be observed by the operator at runtime. This kind of a stochastic event can be used to simulate a disaster response scenario, where a group of robots with pre-computed plans are sent to search and rescue victims at several locations (i.e., the corners). For each location, the robots must cooperate and work together (i.e., meeting at the corner). As more information (e.g., messages reported by the people nearby) is collected at the base station, one of the locations may becomes more likely to have victims. Thus, the operators should guide the robots to search that location and rescue the victims there.

A.2 Cooperative Box-Pushing

In this problem, as shown in Figure 3(b), there are two robots R1 and R2 in a 3×3 grid trying to push the large box (LB) together or independently push the small boxes (SB). Each robot can *turn left*, *turn right*, *move forward*, or *stay* so there are 5 actions per robot. For each action, with probability 0.9, they can turn to the desired direction or move forward and with probability 0.1 they just stay in the same position and orientation. Each robot has 5 observations to identify the object in front, which can be either *an empty field*, *a wall*, *the other robot*, *a small box*, or *the large box*. For each robot, executing an action has a cost of 0.1 for energy consumption. If a robot bumps into a wall, the other robot, or a box without pushing it, it gets a penalty of 5. The standard reward function is designed to encourage cooperation. Specifically, the reward for cooperatively pushing the large box is 100, while the reward of pushing a small box is just 10. Each run includes 100 steps. Once a box is pushed to its goal location, the robots are reset to an initial state.

The high-level commands $C = \{c_0, c_1, c_2, c_3\}$ are designed as follow: (c_0) the robots should push any box; (c_1) the robots should only push the small box on the left side; (c_2) the robots should only push the small box on the right side; (c_3) the robots should only push the large box in the middle. Specifying the corresponding reward function is straightforward. For c_0, we use the original reward function. For c_i ($1 \le i \le 3$), we reward the agents ($+100$) for pushing the right box and penalize them for pushing other boxes (-100).

Similar to the previous domain, we also simulate a random event representing a trapped animal in one of the cells labeled with numbers in Figure 3(b). The animal is hidden behind a box so the robots cannot see it with their cameras. However, if the robots push a box while an animal is on the other side of that box, it will get injured and the robots get a high penalty of 100. The random event is modeled by a FSM with 5 states where state 0 represents no animal and state i ($1 \le i \le 4$) means an animal is trapped in the cell labeled i. If the animal gets injured, the FSM transitions to another state based on a predefined transition function. Again, this event is neither captured by the agents' model nor their policies. The animal can only be observed by the operators during execution time with an additional camera attached behind the boxes. This setting allows us to simulate a scenario where the operators supervise robots performing risk-sensitive tasks. For example, robots doing construction work on a crowded street.