ABSTRACT

Current approaches that model dynamism in DCOPs solve a sequence of static problems, reacting to changes in the environment as the agents observe them. Such approaches thus allow possible predictions on future changes. To overcome this limitation, we introduce Proactive Dynamic DCOPs (PD-DCOPs), a novel formalism to model dynamic DCOPs in the presence of exogenous uncertainty. In contrast to reactive approaches, PD-DCOPs are able to explicitly model the possible changes to the problem, and take such information into account proactively, when solving the dynamically changing problem. The additional expressivity of this formalism allows it to model a wider variety of distributed optimization problems. Our work presents both theoretical and practical contributions that advance current dynamic DCOP models: (i) we introduce the PD-DCOP model, which explicitly captures dynamic changes of the DCOP over time; (ii) we discuss the complexity of this new class of DCOPs; and (iii) we develop both exact and approximation algorithms with quality guarantees to solve PD-DCOPs proactively.

Keywords

Distributed Constraint Optimization; DCOP; Dynamic DCOP

1. INTRODUCTION

Distributed Constraint Optimization Problems (DCOPs) are problems where agents need to coordinate their value assignments to maximize the sum of the resulting constraint utilities [17, 34]. DCOPs have emerged as one of the prominent multi-agent architectures to govern the agents’ autonomous behavior in distributed optimization problems. The model represents a powerful approach to the description and solution of many practical problems, serving several applications such as distributed scheduling, coordination of unmanned air vehicles, smart grid electricity networks, and sensor networks [15, 11, 32, 13, 16, 3, 7, 36]. In many distributed problems of interest, agents interact in complex, uncertain, and dynamic environments. For example, in distributed meeting scheduling, participants could change their preferences and priorities over time. In disaster management, new information (e.g., weather forecasts, priorities on buildings to evacuate) typically becomes available in an incremental manner. Thus, the information flow modifies the environment over time. Unfortunately, the classical DCOP paradigm is unable to model problems that change over time.

Consequently, researchers have introduced Dynamic DCOPs (D-DCOPs) [23, 24, 12, 33], where utility functions can change during the problem-solving process. These models make the common assumption that information on how the problem might change is unavailable. As such, existing approaches react to the changes in the problem and solve the current problem at hand. However, in several applications, the information on how the problem might change is indeed available, or predictable, within some degree of uncertainty. We provide one such example, a distributed meeting scheduling problem, as our motivating example in Section 3.

Therefore, in this paper, (i) we introduce Proactive Dynamic DCOPs (PD-DCOPs), which explicitly model how the DCOP will change over time; (ii) we discuss the complexity of this new class of DCOPs; and (iii) we develop exact and approximation algorithms with quality guarantees to solve PD-DCOPs proactively.

2. BACKGROUND

We now provide background on the regular and dynamic DCOPs as well as the regular and super-stabilizing DPOP algorithms.

2.1 DCOPs

A Distributed Constraint Optimization Problem (DCOP) is a tuple \((\mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha)\), where \(\mathbf{A} = \{a_i\}_{i=1}^p\) is a set of agents; \(\mathbf{X} = \{x_i\}_{i=1}^m\) is a set of decision variables; \(\mathbf{D} = \{D_x\}_{x \in \mathbf{X}}\) is a set of finite domains and each variable \(x \in \mathbf{X}\) takes values from the set \(D_x \in \mathbf{D}\); \(\mathbf{F} = \{f_i\}_{i=1}^m\) is a set of utility functions, each defined over a set of decision variables: \(f_i : X_{x \in \mathbf{X}} \rightarrow \mathbb{R}^+ \cup \{\bot\}\), where \(x_i \subseteq \mathbf{X}\) is scope of \(f_i\) and \(\bot\) is a special element used to denote that a given combination of values for the variables in \(x_i\) is not allowed; and \(\alpha : \mathbf{X} \rightarrow \mathbf{A}\) is a function that associates each decision variable to one agent.

A solution \(\sigma\) is a value assignment for a set \(x_\sigma \subseteq \mathbf{X}\) of variables that is consistent with their respective domains. The utility \(F(\sigma) = \sum_{f_i \in \mathbf{F}} f_i(\sigma)\) is the sum of the utilities across all the applicable utility functions in \(\sigma\). A solution \(\sigma\) is complete if \(x_\sigma = \mathbf{X}\). The goal is to find an optimal complete solution \(x^* = \arg\max_{\sigma} F(\sigma)\).

Given a DCOP \(P, G = (\mathbf{X}, E)\) is the constraint graph of \(P\), where \(\{x, y\} \in E\) if and only if \(\exists f_i \in \mathbf{F}\) such that \(\{x, y\} = x_i\). A DFS
pseudo-tree arrangement for $G$ is a spanning tree $T = (X, E_T)$ of $G$ such that if $f_i \in F$ and $\{x, y\} \subseteq x^{fi}$, then $x$ and $y$ appear in the same branch of $T$. We use $N(a_i) = \{a_j \in A \mid \{x_i, x_j\} \in E\}$ to denote the neighbors of agent $a_i$.

2.2 Dynamic DCOP

A Dynamic DCOP (D-DCOP) is defined as a sequence of DCOPs with changes between them, without an explicit model for how the DCOP will change over time. Solving a D-DCOP optimally means finding a utility-maximal solution for each DCOP in the sequence. Therefore, this approach is reactive since it does not consider future changes. Its advantage is that solving a D-DCOP is no harder than solving $h$ DCOPs, where $h$ is the horizon of the problem. Researchers have used this approach to solve D-DCOPs, where they introduce search- and inference-based approaches that are able to reuse information from previous DCOPs to speed up the search for the solution for the current DCOP [23, 33]. Alternatively, a proactive approach predicts future changes in the D-DCOP and finds robust solutions that require little or no changes despite future changes.

Researchers have also proposed other models for D-DCOPs including a model where agents have deadlines to choose their values [24], a model where agents can have imperfect knowledge about their environment [12], and a model where changes in the constraint graph depends on the value assignments of agents [36].

2.3 DPOP and S-DPOP

The Distributed Pseudo-tree Optimization Procedure (DPOP) [22] is a complete inference algorithm composed of three phases:

- **Pseudo-tree Generation:** The agents build a pseudo-tree [8].
- **UTIL Propagation:** Each agent, starting from the leaves of the pseudo-tree, computes the optimal sum of utilities in its subtree for each value combination of variables in its separator. It does so by adding the utilities of its functions with the variables in its separator and the utilities in the UTIL messages received from its children agents, and projecting out its own variables by optimizing over them.
- **VALUE Propagation:** Each agent, starting from the pseudo-tree root, determines the optimal value for its variables. The root agent does so by choosing the values of its variables from its UTIL computations.

Super-stabilizing DPOP (S-DPOP) [23] is a self-stabilizing extension of DPOP, where the agents restart the DPOP phases when they detect changes in the problem. S-DPOP makes use of information that is not affected by the changes in the problem.

3. MOTIVATING DOMAIN

We now introduce a distributed dynamic meeting scheduling problem, which will serve as a representative domain to motivate our work. In a distributed meeting scheduling problem [15], a set of M weekly meetings need to be scheduled between members of an organization (e.g., employees of a company; students, faculty members, and staff of a university), taking restrictions in their availability as well as their time and location preferences into account. For a meeting $m \in M$, $A_m$ is the set of attendees to the meeting, $s_m$ is the start time of the meeting, $d_m$ is its duration, and $l_m$ is its location. Typically, the attendees commit to a meeting time. However, certain exogenous factors may affect the meeting time preferences of some participants. For instance, depending on the location and time of the meeting, traffic conditions may cause delays that should be taken into account, as they could cause cascading effects.

In a typical DCOP formulation, meeting starting times are modeled as decision variables of the agents. Thus, for a given meeting $m$, each agent $a \in A_m$, controls a pair of decision variables, $x_{sm}$, $x_{dm}$, whose values represent possible start times $s_m$ and locations $l_m$ for the meeting $m$. All agents participating in meeting $m$ need to agree upon a given meeting time and location. This condition can be modeled by imposing an equality constraints on the values of the variables $x_{sm}$ and $x_{dm}$ controlled by the agents in $A_m$. Agent preferences on the time and location for a given meeting can be modeled as unary constraints involving the variable describing the desired meeting. The goal is to find a feasible weekly schedule that maximizes the utility over all attendees.

Typical weekly meeting schedules may be adjusted based on changes of the meeting participants’ preferences. To address such a requirement, one can use a Dynamic DCOP formulation, where a new DCOP problem, representing the scheduling problem for a single week, can be modeled according to the previous formulation and solved, as soon as some agent’s meeting preference changes. However, these formulations exhibit several limitations: (i) they fail to capture the presence of exogenous factors (e.g., traffic conditions) in the dynamic aspect of the problem, and (ii) they do not take account the inconvenience of the participants to change their schedule, when the preference of some agents are updated. Our proposed PD-DCOP model alleviates these limitations by acting proactively during the problem resolution, which allows us to make a step forward towards more refined dynamic solutions.

4. PD-DCOP MODEL

A Proactive Dynamic DCOP (PD-DCOP) is a tuple $(A, X, D, F, h, T, c, \gamma, p, \alpha, \Omega)$, where:

- **$A = \{a_1\}^p_{i=1}$ is a set of agents.**
- **$X = \{x_i\}^m_{i=1}$ is a mixed set of decision and random variables.** To differentiate between decision variables and random variables, we use $Y \subseteq X$ to denote the set of random variables that model uncontrollable stochastic events (e.g., traffic, weather, malfunctioning devices).
- **$D = \{D_x\}_{x \in X}$ is a set of finite domains.** Each variable $x \in X$ takes values from the set $D_x \in D$. We also use $\Omega = \{\Omega_y\}_{y \in Y} \subseteq D$ to denote the set of event spaces for the random variables (e.g., different traffic conditions, weather conditions, or stress levels to which a device is subjected) to such that each $y \in Y$ takes values in $\Omega_y$.
- **$F = \{f_i\}^m_{i=1}$ is a set of reward functions, each defined over a mixed set of decision variables and random variables: $f_i : \chi_{x \in X} \to \mathbb{R}^+ \cup \{\bot\}$, where $\chi^{fi} \subseteq X$ is scope of $f_i$ and $\bot$ is a special element used to denote that a given combination of values for the variables in $\chi^{fi}$ is not allowed.**
- **$h \in \mathbb{N}$ is a finite horizon in which the agents can change the values of their variables.**
- **$T = \{T_y\}_{y \in Y}$ is the set of transition functions $T_y : \Omega_y \times \Omega_y \to [0, 1] \subseteq \mathbb{R}$ for the random variables $y \in Y$, describing the probability for a random variable to change its value in successive time steps. For a time step $t > 0$, and values $\omega_i \in \Omega_y$, $\omega_j \in \Omega_y$, $T_y(\omega_i, \omega_j) = P(y^t = \omega_j \mid y^{t-1} = \omega_i)$, where $y^t$ denotes the value of the variable $y$ at time step $t$, and $P$ is a probability measure. Thus, $T_y(\omega_i, \omega_j)$ describes the probability for the random variable $y$ to change its value from $\omega_i$ at a time step $t-1$ to $\omega_j$ at a time step $t$. Finally, $\sum_{\omega_j \in \Omega_y} T_y(\omega_i, \omega_j) = 1$ for all $\omega_i \in \Omega_y$.**
- **$c \in \mathbb{R}^+$ is a switching cost, which is the cost associated with the change in the value of a decision variable from one time step to
The goal of a PD-DCOP is to find a sequence of \( h+1 \) assignments \( x^* \) for all the decision variables in \( X \setminus Y \):

\[
x^* = \arg\max_{x \in \{x_1, \ldots, x_h\}^{\Sigma^h+1}} F^h(x)
\]

\[
F^h(x) = \sum_{t=0}^{h-1} \gamma^t \left[ F^t_x(x^*) + F^t_y(x^*) \right]
\]

where \( \gamma = 0, 1 \) is a discount factor, which represents the decrease in the importance of rewards/costs over time.

\[ p^h_Y = \{ p^h_Y(y) \mid y \in Y \} \text{ is a set of initial probability distributions for the random variables } y \in Y. \]

\[ \alpha : X \setminus Y \rightarrow A \text{ is a function that associates each decision variable to one agent. We assume that the random variables are not under the control of the agents and are independent of decision variables. Thus, their values are solely determined according to their transition functions.} \]

Throughout this paper, we refer to decision (resp. random) variables as with the letter \( x \) (resp. \( y \)). We also assume that each agent controls exactly one decision variable (thus, it is not a bijection), and that each reward function \( f_i \in F \) associates with at most one random variable \( y_i).^2 \]

The goal of a PD-DCOP is to find a sequence of \( h + 1 \) assignments \( x^* \) for all the decision variables in \( X \setminus Y \):

\[
x^* = \arg\max_{x \in \{x_1, \ldots, x_h\}^{\Sigma^h+1}} F^h(x)
\]

The goal of a PD-DCOP is to find an assignment of values to its decision variables that maximizes the sum of two terms. The first term maximizes the discounted net utility, that is, the discounted rewards for the functions that do not involve exogenous factors (\( F_x \)) and the expected discounted random rewards (\( F_y \)) minus the discounted penalties over the first \( h \) time steps. The second term maximizes the discounted future rewards for the problem.

While the PD-DCOP model can be used to capture the presence of exogenous factors in the dynamic aspect of the problem, it can also model dynamic changes to the DCOP constraint graph, through the transition functions. In particular, the deletion of a constraint will force the random variable associated with that constraint to transit to a 0 reward value for all decision variables; the addition of a constraint can be handled by defining a 0 reward constraint in the model from the start, and updating its reward when the constraint is added.

### 4.1 Modeling the Motivating Domain

PD-DCOPs can naturally handle the dynamic characteristic of the distributed dynamic meeting scheduling problem that we use as our motivating domain. Uncontrollable events, such as traffic conditions, affecting the meeting times of agents can be modeled via random variables. In particular, in our model, each pair of agent’s variables \( x_{s,m} \) and \( x_{t,m} \), describing the time and location of a meeting \( m \), is associated to a random variable \( y_m \in Y \) describing the different traffic conditions that can affect agents’ time and location preferences for the meeting. Traffic predictions are often available and they are modeled via the transition functions. Additionally, rescheduling meetings is inconvenient for the meeting participants and this inconvenience is modeled via switching costs for each attendee, imposed when a meeting is forced to be rescheduled. Finally, the PD-DCOP horizon captures the horizon (i.e., number of weeks) of the scheduling problem (e.g., the DCOP problem of each time step corresponds to the scheduling problem for a single week).

#### 4.2 Theoretical Properties

We now describe some of the theoretical properties of PD-DCOPs.

**Theorem 1.** Solving a PD-DCOP is PSPACE-complete (-hard) if the horizon \( h \) is polynomial (exponential) in \( |X| \).

**Proof:** We first consider the case when \( h \) is polynomial in \( |X| \). Membership in PSPACE follows from the existence of a naive depth-first search to solve PD-DCOPs, where a non-deterministic branch is opened for each complete assignment of the PD-DCOP’s decision variables and for each time step \( 0 \leq t \leq h \). The algorithm requires linear space in the number of variables and horizon length. We reduce the satisfiability of quantified Boolean formula (QSAT) to a PD-DCOP with \( 0 \) horizon. Each existential Boolean variable in the QSAT is mapped to a corresponding decision variable in the PD-DCOP, and each universal Boolean variable in the QSAT is mapped to a PD-DCOP random variable. The domains \( D_x \) of all variables \( x \in X \) are the sets of values \( \{0, 1\} \), corresponding respectively to the evaluations, false and true, of the QSAT variables. The initial probability distribution \( p^0_y \) of each PD-DCOP random variable \( y \in Y \) is set to be the uniform distribution. Each QSAT clause \( c \) is mapped to a PD-DCOP reward function \( f_c \), whose scope involves all and only the PD-DCOP-corresponding boolean variables appearing in \( c \), and such that:

\[
f_c(x^c) = \begin{cases} 1, & \text{if } c(x^c) = \text{true} \\ \bot, & \text{otherwise.} \end{cases}
\]

where \( c(x^c) \) denotes the instantiation of the values of the variables in \( x^c \) to the truth values of the corresponding literals of \( c \). In other
words, a clause is satisfied iff the equivalent reward function preserves its semantics. The choices for, the switching cost, the discount factor \( \gamma \), and the transition function \( T_x \), for each \( y \in Y \), of the PD-DCOP, are immaterial. The reduction is linear in the size of the original quantified Boolean formula. The quantified Boolean formula is satisfiable iff the equivalent PD-DCOP has at least one solution \( x \) whose cost \( F(x) \neq \perp \).

We next consider the case when \( h \) is exponential in \( X \). In such case solving PD-DCOPs is PSPACE-hard as storing a solution requires space exponential in \( |X| \).

**Absolute Error Bound:** Let \( U^\infty \) denote the optimal solution quality with an infinite horizon and \( U^h \) denote the optimal solution quality with a finite horizon \( h \). Thus, \( \epsilon \geq U^\infty - U^h \) and we first describe this error bound. Let's define \( F^\Delta = \max_{y \in U} (F(x^* \cup y) - F(x \cup y)) \), where \( x \) and \( y \) are assignments of values to all decision and random variables, respectively; \( F \) is the overall reward function for a (regular) DCOP in the PD-DCOP; and \( x^* \) is an optimal value assignment of decision variables for that regular DCOP given value assignment \( y \).

Notice that the quantity \( \Delta(x^*_t) + \Delta(x^*_{t+1}) - \Delta(x^*_t) \) is the utility difference between the value assignment \( x^*_t \) and \( x^*_{t+1} \) for the regular DCOPs in time step \( t \). So, we have:

\[
\Delta(x^*_t) + \Delta(x^*_{t+1}) - \Delta(x^*_t) \leq F^\Delta
\]

which concludes the proof.

**Corollary 1.** Given a maximum acceptable absolute error \( \epsilon \), the minimum horizon \( h \) is \( \log_{1-\epsilon} \frac{U^\infty}{\epsilon} \).

**Upper Bound on Optimal Quality:** We now describe an upper bound on the optimal solution quality \( F^h(x^*) \). Let \( x^* = (x^*_0, \ldots, x^*_n) \) be the vector of assignments, where:

\[
\begin{align*}
\hat{x}^*_t &= \max_{x \in X} \gamma^t \left[ F^h(x) + F^h(x) \right] \text{ if } 0 \leq t < h \\
\hat{x}^*_t &= \max_{x \in X} \gamma^t \left[ F^h(x) + F^h(x) \right] \text{ otherwise }
\end{align*}
\]

and \( \hat{F}^h(x) = \sum_{t=0}^{h-1} \gamma^t \left[ F^h(x) + F^h(x) \right] + \hat{F}(x) + \hat{F}(x) \).

**Theorem 3.** \( \forall x \in \Sigma^{h+1} : F^h(x) \leq \hat{F}^h(x^*) \).

**Proof:** For any given assignment \( x \in \Sigma^{h+1} \), \( F^h(x) \) is a clear lower bound for \( \hat{F}^h(x^*) \).

For the upper bound, let \( \hat{F}^h(x) \) be the \( h \)-th component of the \( \hat{F}^h(\cdot) \), defined as:

\[
\hat{F}^h(x)_t = \left\{ \begin{array}{ll}
\gamma^t [F^h(x_t) + F^h(x_t)] & \text{if } 0 \leq t < h \\
F^h(x_t) + F^h(x_t) & \text{otherwise }
\end{array} \right.
\]

with \( x_t \), defined as the \( t \)-th value assignment in the PD-DCOP solution \( x \). Analogously, let us denote with \( \hat{F}^h(\cdot) \) the \( t \)-th component of the \( \hat{F}^h(\cdot) \), defined as:

\[
\hat{F}^h(x)_t = \left\{ \begin{array}{ll}
\gamma^t [F^h(x_t) + F^h(x_t)] & \text{if } 0 \leq t < h \\
F^h(x_t) + F^h(x_t) & \text{otherwise }
\end{array} \right.
\]

It follows that for all \( 0 \leq t < h \):

\[
\hat{F}^h(x^*_t) = \gamma^t [F^h(x^*_t) + F^h(x^*_t)] - [\gamma^t [F^h(x^*_t) + F^h(x^*_t)]]
\]

\[
\leq \max_{x \in \Sigma} \gamma^t [F^h(x) + F^h(x)] = \hat{F}^h(x^*_t)
\]

where \( x^*_t \) (resp. \( x^*_t \)) is the \( t \)-th component of the PD-DCOP solution vector \( x^* \) (resp. \( x^* \)). For \( t = h \), it follows:

\[
\hat{F}^h(x^*_h) = \hat{F}(x^*_h) + \hat{F}(x^*_h)
\]

\[
\leq \max_{x \in \Sigma} \left[ \hat{F}(x) + \hat{F}(x) \right] = \hat{F}^h(x^*_h)
\]

Thus, from the two inequalities above, it follows:

\[
\hat{F}^h(x^*) \leq \sum_{t=0}^{h} \hat{F}^h(x^*_t) = \hat{F}^h(x^*)
\]

which concludes the proof.

**Corollary 2.** The approximation ratio \( \rho \) is \( \frac{\hat{F}^h(x^*)}{F^h(x^*)} \) for any solution \( x \).
5. PD-DCOP ALGORITHMS

We now introduce exact and approximation PD-DCOP algorithms.

5.1 Exact Approach

We first propose an exact approach, which transforms a PD-DCOP into an equivalent DCOP and solves it using any off-the-shelf DCOP algorithm. Since the transition of random variables is independent of the assignment of values to decision variables, this problem can be viewed as a Markov chain. Thus, it is possible to collapse an entire PD-DCOP into a single DCOP, where (1) each reward function \( h_i \) in this new DCOP captures the sum of rewards of the reward function \( f_i \in F \) across all time steps, and (2) the domain of each decision variable is the set of all possible combination of values of that decision variable across all time steps. However, this process needs to be done in a distributed manner.

We divide the reward functions into two types: (1) The functions \( f_i \in F \) whose scope \( x_i \cap Y = \emptyset \) includes exclusively decision variables, and (2) the functions \( f_i \in F \) whose scope \( x_i \cap Y \neq \emptyset \) includes one random variable. In both cases, let \( x_i = (x_{i,0}, \ldots, x_{i,h}) \) denote the vector of value assignments to all decision variables in \( x_i \) for each time step.

Then, each function \( f_i \in F \) whose scope includes only decision variables can be replaced by a function \( f_i \):

\[
F_i(x_i) = \sum_{t=0}^{h-1} F_i^t(x_{i,t}) + F_i^h(x_{i,h})
\]

where the first term with the summation is the reward for the first \( h \) time steps and the second term is the reward for the remaining time steps. Each function \( f_i \in F \) whose scope includes random variables can be replaced by a unary function \( G_i \):

\[
G_i(x_i) = \sum_{t=0}^{h-1} \gamma^t \sum_{\omega \in \Omega_i} F_i^t(x_{i,t}) \cdot p_{i,t}^\omega(\omega)
\]

where the first term (Equation (14)) is the reward for the first \( h \) time steps and the second term (Equation (15)) is the reward for the remaining time steps. The function \( f_i \) is recursively defined according to Equation (10). Additionally, each decision variable \( x_i \) will have a unary function \( C_i \):

\[
C_i(x_i) = \sum_{t=0}^{h-1} \gamma^t \cdot \Delta(x_{i,t}, x_{i,t+1})
\]

Algorithm 1: LOCAL SEARCH()

1. \( \text{iter} \leftarrow 1 \)
2. \( (v_0^0, v_1^1, \ldots, v_h^h) \leftarrow (\text{Null}, \text{Null}, \ldots, \text{Null}) \)
3. \( (v_0^0, v_1^1, \ldots, v_h^h) \leftarrow \text{INITIAL_ASSIGNMENT}() \)
4. \( \text{context} \leftarrow ((x_j, t, \text{Null} | x_j \in N(a_i), 0 \leq t \leq h)) \)
5. Send VALUE\((v_0^h, v_1^h, \ldots, v_h^h)\) to all neighbors

Procedure CalcGain()

6. \( (u_0^0, u_1^1, \ldots, u_h^h) \leftarrow \text{CALCULATE}(v_0^0, v_1^1, \ldots, v_h^h) \)
7. \( u^* \leftarrow -\infty \)
8. foreach \( (d_0^0, d_1^1, \ldots, d_h^h) \in x_i \cap D, \omega \) do
9. \( u \leftarrow \text{CALCULATE}(d_0^0, d_1^1, \ldots, d_h^h) \)
10. if \( u > u^* \) then \( u^* \leftarrow u \)
11. \( (v_0^0, v_1^1, \ldots, v_h^h) \leftarrow (d_0^0, d_1^1, \ldots, d_h^h) \)
12. if \( u^* \neq -\infty \) then
13. \( (u_0^0, u_1^1, \ldots, u_h^h) \leftarrow \text{CALCULATE}(v_0^0, v_1^1, \ldots, v_h^h) \)
14. \( (u_0^0, u_1^1, \ldots, u_h^h) \leftarrow (u_0^0, u_1^1, \ldots, u_h^h) - (v_0^0, v_1^1, \ldots, v_h^h) \)
15. \( (u_0^0, u_1^1, \ldots, u_h^h) \leftarrow (u_0^0, u_1^1, \ldots, u_h^h) \)
16. else
17. \( (v_0^0, v_1^1, \ldots, v_h^h) \leftarrow (\text{Null}, \text{Null}, \ldots, \text{Null}) \)
18. Send GAIN\((v_0^h, v_1^h, \ldots, v_h^h)\) to all neighbors

changing environments. Algorithm 1 shows its pseudocode, where each agent \( a_i \) maintains the following data structures:

- \( \text{iter} \) is the current iteration number.
- \( \text{context} \) is a vector of tuples \((x_j, t, \text{Null})\) for all its neighboring variables \( x_j \in N(a_i) \). Each of these tuples represents the agent’s assumption that variable \( x_j \) is assigned value \( v_j^t \) at time step \( t \).
- \((v_0^0, v_1^1, \ldots, v_h^h)\) is a vector of the agent’s current value assignment for its variable \( x_i \) at each time step.
- \((u_0^0, u_1^1, \ldots, u_h^h)\) is a vector of the agent’s best value assignment for its variable \( x_i \) at each time step.
- \((v_0^0, v_1^1, \ldots, v_h^h)\) is a vector of the agent’s current utility (rewards from reward functions minus costs from switching costs) given its current value assignment at each time step.
- \((u_0^0, u_1^1, \ldots, u_h^h)\) is a vector of the agent’s best utility (rewards from reward functions minus costs from switching costs) given its current value assignment at each time step.
- \((v_0^0, v_1^1, \ldots, v_h^h)\) is a vector of the agent’s current value assignment for its variable \( x_i \) at each time step.
- \((u_0^0, u_1^1, \ldots, u_h^h)\) is a vector of the agent’s best value assignment for its variable \( x_i \) at each time step.
- \((v_0^0, v_1^1, \ldots, v_h^h)\) is a vector of the agent’s current utility (rewards from reward functions minus costs from switching costs) given its current value assignment at each time step.
- \((u_0^0, u_1^1, \ldots, u_h^h)\) is a vector of the agent’s best utility (rewards from reward functions minus costs from switching costs) given its current value assignment at each time step.

The high-level ideas are as follows: (1) Each agent \( a_i \) starts by finding an initial value assignment to its variable \( x_i \) for each time step \( 0 \leq t \leq h \) and initializes its context \( \text{context} \). (2) Each agent uses VALUE messages to ensure that it has the correct assumption on its neighboring agents’ variables’ values. (3) Each agent computes its current utilities given its current value assignments, its best utilities over all possible value assignments, and its best gain in utilities, and sends this gain in a GAIN message to all its neighbors. (4) Each agent changes the value of its variable for time step \( t \) if its gain for that time step is the largest over all its neighbors’ gain for that time step, and repeats steps 2 through 4 until a termination condition is met. In more detail:

Step 1: Each agent initializes its vector of best values to a vector of Nulls (line 2) and calls INITIAL_ASSIGNMENT to initializes its current values (line 3). The values can be initialized randomly or according to some heuristic function. We describe later one such heuristic. Finally, the agent initializes its context, where it assumes that the values for its neighbors is null for all time steps (line 4).

Step 2: The agent sends its current value assignment in a VALUE message to all neighbors (line 5). When it receives a VALUE message from its neighbor, it updates its context with the value assignments in that message (lines 19-21). When it has received VALUE
messages from all neighbors in the current iteration, it means that its context now correctly reflects the neighbors’ actual values. It then calls CALCAGAIN to start Step 3 (line 23).

**Step 3:** In the CALCAGAIN procedure, the agent calls CALCUTILS to calculate its utility for each time step given its current value assignments and its neighbors’ current value assignments recorded in its context (line 6). The utility for a time step \( t \) is made out of two components (line 41). The first component is the sum of rewards over all reward functions that involve the agent, under the assumption that the agent takes on its current value and its neighbors take on their values according to its context. Specifically, if the scope of the reward function \( F_j \) involves \( v_i^t \), then \( F_j(v_i^t, v_j^t) \) is a function of the agent’s current value \( v_i^t \) and its neighbor’s value \( v_j^t \) in its context and is defined according to Equations (11) to (12). If the scope involves both decision and random variables, then \( F_j(v_i^t, v_j^t) \) is a unary constraint that is only a function of the agent’s current value \( v_i^t \) and is defined according to Equations (13) to (15). The second component is the cost of switching values from the previous time step \( t - 1 \) to the current time step \( t \) and switching from the current time step to the next time step \( t + 1 \). This cost is \( c \) if the values in two subsequent time steps are different and 0 otherwise. The variable \( c_i^t \) captures this cost (lines 35-40). The (net) utility is thus the reward from the reward functions minus the switching cost (line 41).

The agent then searches over all possible combination of values for its variable across all time steps to find the best value assignment that results in the largest cumulative cost across all time steps (lines 8-12). It then computes the net gain in utility at each time step by subtracting the utility of the best value assignment with the utility of the current value assignment (lines 13-15).

**Step 4:** The agent sends its gains in a GAIN message to all neighbors (line 18). When it receives a GAIN message from its neighbor, it updates its best value \( v_i^t \) for time step \( t \) to null if its gain is non-positive (i.e., \( u_i^t \leq 0 \)) or its neighbor has a larger gain (i.e., \( u_i^t > u_j^t \)) for that same time step (line 27). When it has received GAIN messages from all neighbors in the current iteration, it means that it has identified, for each time step, whether its gain is the largest over all its neighbors’ gains. The time steps where it has the largest gain are exactly those time steps where \( v_i^t \) is not null. The agent thus assigns its best value for these time steps as its current value and restarts Step 2 by sending a VALUE message that contains its new values to all its neighbors (lines 29-33).

---

**Procedure When Receive VALUE\((v_i^0, v_i^1, \ldots, v_i^h)\)**

- **foreach** \( t \) from 0 to \( h \) do
  - if \( v_i^t \neq \text{Null} \) then
  - Update \((x_s, t, v_i^t) \in \text{context} \) with \((x_s, t, v_i^t)\)
- if received VALUE messages from all neighbors in this iteration then
  - CALCAGAIN
- \( \text{iter} \leftarrow \text{iter} + 1 \)

**Procedure When Receive GAIN\((\hat{u}_i^0, \hat{u}_i^1, \ldots, \hat{u}_i^h)\)**

- if \((\hat{u}_i^0, \hat{u}_i^1, \ldots, \hat{u}_i^h) \neq (\text{Null}, \text{Null}, \ldots, \text{Null})\) then
  - foreach \( t \) from 0 to \( h \) do
    - if \( u_i^t \leq 0 \) or \( \hat{u}_i^t > u_i^t \) then
      - \( v_i^t \leftarrow \text{Null} \)
- if received GAIN messages from all neighbors in this iteration then
  - foreach \( t \) from 0 to \( h \) do
    - if \( u_i^t \neq \text{Null} \) then
      - Send VALUE\((v_i^t, \hat{u}_i^t, \ldots, v_i^h)\) to all neighbors

---

**Function CalcUtils\((v_i^0, v_i^1, \ldots, v_i^h)\)**

- **foreach** \( t \) from 0 to \( h \) do
  - if \( t = 0 \) then
    - \( c_i^t \leftarrow \gamma^0 \cdot \text{cost}(v_i^0, v_i^1) \)
  - else if \( t = h \) then
    - \( c_i^t \leftarrow \gamma^h \cdot \text{cost}(v_i^h, v_i^h) \)
  - else
    - \( c_i^t \leftarrow \gamma^t \cdot \text{cost}(v_i^t, v_i^{t+1}) \)
  - \( u_i^t \leftarrow \sum_{j \in i} F_j(v_i^t) \)
- return \( c_i^0 + c_i^h + 1 \)

**Function CalcCumulativeUtil\((v_i^0, v_i^1, \ldots, v_i^h)\)**

- \( c \leftarrow 0 \)
- foreach \( t \) from 0 to \( h - 1 \) do
  - \( c \leftarrow c + \gamma^t \cdot \text{cost}(v_i^t, v_i^{t+1}) \)
- return \( c \)
Theorem 4. The agent's space required by the PD-DCOP approximation approach is \(O(\mathcal{L} + (h + 1)|A|)\).

Proof: In our approximation algorithm, each agent first calls the \textsc{InitialAssignment} function to find an initial value assignment to its variable for each time step \(0 \leq t \leq h\) (line 3). Thus, the memory requirement of this step is \(O((h + 1) + \mathcal{L})\) at each agent. Next, each agent running our approximation algorithm performs a local search step (lines 4-5), which is analogous to that of MGM. However, different from MGM, our agents search for tuples of \(h + 1\) values, one for each time step in the horizon. Thus, at each iteration, and for each time step \(t\), each agent stores a vector of values for its current and best variable assignments for its variable; a vector of the agent's utilities and best utilities given its current value assignments; and a vector of the agent's best gain in utility. In addition, each agent stores the context of its neighbors' values for each time step \(t\), which requires \(O((h + 1) \cdot |N(a_i)|)\) space. Thus, the overall space requirement for our approximation algorithm is \(O(\mathcal{L} + (h + 1)|A|)\) per agent.

In Section 7, we present two approximation approaches: LS-DPOP, which uses DPOP as the \textsc{InitialAssignment} function, and LS-RAND, which uses a random assignment function. Thus, the space complexity of LS-DPOP is \(O(d^{n^*})\) – where \(d\) is the size of the largest domain and \(n^*\) is the induced width of the pseudo-tree – as DPOP dominates the space complexity in the initialization step. The space complexity of LS-RAND is instead dominated by the second step of the approximation PD-DCOP algorithm, and it is thus \(O((h + 1)|A|)\).

6. RELATED WORK

Aside from the D-DCOPs described in Sections 1 and 2, several approaches have been proposed to proactively solve centralized Dynamic CSPs, where value assignments of variables or utilities of constraints may change according to some probabilistic model [29, 10]. The goal is typically to find a solution that is robust to possible changes. Other related models include Mixed CSPs [6], which model decision problems under uncertainty by introducing state variables, which are not under control of the solver, and seek assignments that are consistent to any state of the world; and Stochastic CSPs [30, 27], which introduce probability distributions that are associated to outcomes of state variables, and seek solutions that maximize the probability of constraint consistencies. While these proactive approaches have been used to solve CSP variants, they have not been used to solve Dynamic DCOPs to the best of our knowledge.

Researchers have also introduced Markovian D-DCOPs (MD-DCOPs), which model D-DCOPs with state variables that are beyond the control of agents [20]. However, they assume that the state is observable to the agents, while PD-DCOPs assume otherwise. Additionally, MD-DCOP agents do not incur a cost for changing values in MD-DCOPs and only a reactive online learning approach to solving the problem has been proposed thus far.

Another related body of work is Decentralized Markov Decision Processes (Dec-MDPs) [2]. In a Dec-MDP, agents can also observe its local state (the global state is the combination of all local states), and the goal of a Dec-MDP is to find a policy that maps each local state to the action for each agent. Thus, like PD-DCOPs, it too solves a sequential decision making problem. However, Dec-MDPs are typically solved in a centralized manner [2, 1, 4, 5] due to its high complexity – solving Dec-MDPs optimally is NEXP-hard even for the case with only two agents [2]. In contrast, PD-DCOPs are solved in a decentralized manner and its complexity is only PSPACE-hard. The reason for the lower complexity is because the solution of PD-DCOPs are open-loop policies, which are policies that are not dependent on state observations.

Decentralized Partially Observable MDPs (Dec-POMDPs) [2] is a generalization of Dec-MDPs, where an agent may not accurately observe its local state. Instead, it maintains a belief of its local state. A Dec-POMDP policy thus maps each belief to an action for each agent. Solving Dec-POMDPs is also NEXP-hard [2] and they are also typically solved in a centralized manner [9, 26, 25, 31, 5, 21] with some exceptions [18]. Researchers have also developed a hybrid model, called ND-POMDP [19], which is a Dec-POMDP that exploits locality of agent interactions like a DCOP.

In summary, one can view DCOPs and Dec-(PO)MDPs as two ends of a spectrum of offline distributed planning models. In terms of expressiveness, DCOPs can solve single timestep problems while Dec-(PO)MDPs can solve sequential problems. However, DCOPs are NP-hard while Dec-(PO)MDPs are NEXP-hard. PD-DCOPs attempt to balance the trade off between expressiveness and complexity by searching for open-loop policies instead of closed-loop policies of Dec-(PO)MDPs. They are more expressive than DCOPs at the cost of a higher complexity, yet not as expressive as Dec-(PO)MDPs, but also without their prohibitive complexity.

7. EXPERIMENTAL RESULTS

We empirically evaluate Collapsed DPOP (C-DPOP), which collapses the PD-DCOP into a DCOP and solves it with DPOP; Local Search (Random) (LS-RAND), which runs the local search algorithm with random initial values; and Local Search (S-DPOP) (LS-SDPOP), which runs the algorithm with S-DPOP and the \(h_3\) heuristic function to generate pseudo-trees. In contrast to many experiments in the literature, our experiments are performed in an actual distributed system, where each agent is an Intel i7 Quadcore 3.4GHz machine with 16GB of RAM, connected in a local area network. We thus report actual distributed runtimes. We impose a timeout of 30 minutes and a memory limit of 16 GB. Results are averaged over 30 runs. We use the following default configuration: Number of agents and decision variables \(|A| = |X \setminus Y| = 12\); number of random variables \(|Y| = 0.25 \cdot |X \setminus Y|\); domain size \(|D_a| = |\Omega_a| = 3\); horizon \(h = 3\); switching cost \(c = 50\); constraint densities \(p_{1}^{*} = p_{2}^{*} = 0.5\);\(^1\) and constraint tightness \(p_2 = 0.8\).

\(^1\)\(p_{1}^{*}\) is the density of functions between two decision variables, \(p_{1}^{*}\) is the...
Random Networks. We first vary the weight \( w \) of the pseudo-tree construction heuristic \( h_3 \) to identify the best weight for LS-SDPOP. Figure 1(left) shows the runtimes of LS-SDPOP. At \( w = 0 \), the heuristic \( h_3 \) corresponds the max-degree heuristic \( h_2 \) and, at \( w = 1 \), the heuristic is analogous to our \( h_1 \) heuristic. The runtimes are high at both extremes for the following reasons: When \( w = 0 \), LS-SDPOP exploits weakly the reuse of information, and when \( w = 1 \), the resulting pseudo-trees have large depths, which in turn result in large runtimes. The best weight is found at \( w = 0.4 \), thus we use this value for the remaining experiments.

We then vary the switching cost \( c \) of the problem from 0 to 500 to investigate its impact on the algorithms’ performance. Figure 1(center) shows the number of iterations it takes for the local search algorithms to converge from the initial solution. When \( c = 0 \), the initial solution found by LS-SDPOP is an optimal solution since the initial solution already optimizes the utilities of the problem over all time steps ignoring switching costs. Thus, it requires 0 iterations to converge. For sufficiently large costs \( (c \geq 100) \), the optimal solution is one where the values for each agent is the same across all time steps since the cost of changing values is larger than the gain in utility. Thus, the number of iterations they take to converge is the same for all large switching costs. At intermediate cost values \((0 < c < 100)\), they require an increasing number of iterations to converge. Finally, LS-RAND requires more iterations to converge than LS-SDPOP since it starts with poorer initial solutions.

We also vary the horizon \( h \) of the problem from 2 to 10 to evaluate the scalability of the algorithms.\(^4\) Figure 1(right) shows the runtimes of all three algorithms. As expected, the runtimes increase when the horizon increases. When the horizon \( h \geq 6 \) is sufficiently large, LS-SDPOP is faster than LS-RAND indicating that the overhead of finding good initial solutions with S-DPOP is worth the savings in runtime to converge to the final solution.

Finally, we vary the number of agents \( |A| \) (and thus the number of the decision variables) of the problem from 2 to 16. Table 1 tabulates the runtimes and the approximation ratio \( \rho \) for all three algorithms. The runtimes of LS-SDPOP without reusing information are shown in parentheses. C-DPOP times out after \(|A| \geq 8 \). In general, the runtimes of C-DPOP is largest, followed by the runtimes of LS-SDPOP and the runtimes of LS-RAND. The difference in runtimes increases with increasing number of agents, indicating that the overhead to find good initial solutions with S-DPOP is not worth the savings in convergence runtime. As expected, the approximation ratio \( \rho \) with C-DPOP is the smallest, since it finds optimal solutions, whereas the ratios of the local search algorithms are of similar values, indicating that they converge to solutions with similar qualities. Therefore, LS-SDPOP is preferred in problems with few agents but large horizons and LS-RAND is preferred in problems with many agents but small horizons.

However, another factor to consider when choosing which algorithm to run is their memory requirement. LS-SDPOP suffer from the same exponential memory requirement (in the induced width of the pseudo-tree) of DPOP [22]. In contrast, LS-RAND’s memory requirement is only linear in the number of agents and the horizon. Thus, LS-RAND is preferred in problems where agents have a limited amount of memory (e.g., sensor networks).

Dynamic Distributed Meeting Scheduling. We also evaluate our PD-DCA algorithms on the dynamic distributed meeting scheduling problem introduced in Section 3, where we use the following parameters: We allow each meeting to be scheduled in 4 different starting time and 2 locations. We generate the underlying graph topology randomly, using the same settings described in the previous experiments. Thus, the number of meetings and the number of meetings participants are not bounded by any fixed value. In order to ensure that each agent controls exactly one decision variable, we use the pseudo-agent decomposition technique [35]. Inequality constraints between the meeting start time and locations ensure that an agent can attend at most one meeting at a given time, and that no two meetings are held in the same location at the same time. In addition, start times and locations of each meeting’s participants are enforced to be equal, so as to produce feasible schedules. Finally, agents’ preferences on time and meeting locations are modeled through unary costs functions. We use the same heuristic weight, switching costs, and horizon settings from the previous experiment on random networks.

Table 2 illustrates the average runtimes (in ms) and the percentage of feasible solutions over 30 instances, returned by the algorithms at varying the number of agents \( |A| \) from 2 to 12. Similar to the results analyzed for random networks, both approximation approaches (LS-SDPOP and LS-RAND) can produce solutions faster than the exact approach. However, the number of satisfiable instances decreases with increasing number of agents. In particular, the quality of solutions found by LS-SDPOP degrades slower than the quality of solutions found by LS-RAND. The reason is likely because LS-SDPOP starts with a better initial solution than LS-RAND. As expected, these results reveal that for our PD-DCA approximation approach, the initial solution is crucial to ensure convergence to solutions of high quality within a bounded runtime.

8. CONCLUSIONS

In real-world applications, agents often act in dynamic environments. Thus, the Dynamic DCOP formulation is attractive to model such problems. Current research has focused at solving such problems reactively, thus discarding the information on possible future changes, which is often available in many applications. To cope with this limitation, we (i) introduce Proactive Dynamic DCOPs (PD-DCOPs), which model the dynamism in Dynamic DCOPs; (ii) provide theoretical results on the complexity class of PD-DCOPs; and (iii) develop an exact PD-DCOP algorithm that solves the problem proactively as well as an approximation algorithm with quality guarantees that can scale to larger and more complex problems. Finally, in contrast to many experiments in the literature, we evaluate our algorithms on an actual distributed system, which will ease the transition to real-world applications.
REFERENCES


