Learning Optimal Temperature Region for Solving Mixed Integer Functional DCOPs

Saaduddin Mahmud[1] , **Md. Mosaddek Khan**[1] , **Moumita Choudhury**[1] , **Long Tran-Thanh**[2] and **Nicholas R. Jennings**[3]
[1]Department of Computer Science and Engineering, University of Dhaka
[2]School of Electronics and Computer Science, University of Southampton
[3]Departments of Computing and Electrical and Electronic Engineering, Imperial College London
saadmahmud14@gmail.com, mosaddek@du.ac.bd, moumitach22@gmail.com,
ltt08r@ecs.soton.ac.uk, n.jennings@imperial.ac.uk

Abstract

Distributed Constraint Optimization Problems (DCOPs) are an important framework for modeling coordinated decision-making problems in multi-agent systems with a set of discrete variables. Later works have extended DCOPs to model problems with a set of continuous variables, named Functional DCOPs (F-DCOPs). In this paper, we combine both of these frameworks into the Mixed Integer Functional DCOP (MIF-DCOP) framework that can deal with problems regardless of their variables' type. We then propose a novel algorithm – Distributed Parallel Simulated Annealing (DPSA), where agents cooperatively learn the optimal parameter configuration for the algorithm while also solving the given problem using the learned knowledge. Finally, we empirically evaluate our approach in DCOP, F-DCOP, and MIF-DCOP settings and show that DPSA produces solutions of significantly better quality than the state-of-the-art non-exact algorithms in their corresponding settings.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) are a widely used framework for coordinating interactions in cooperative multi-agent systems. More specifically, agents in this framework coordinate value assignments to their variables in such a way that minimizes constraint violations by optimizing their aggregated costs [Yokoo *et al.*, 1998]. DCOPs have gained popularity due to their applications in various real-world multi-agent coordination problems, including distributed meeting scheduling [Maheswaran *et al.*, 2004b], sensor networks [Farinelli *et al.*, 2014] and smart grids [Fioretto *et al.*, 2017].

Over the last two decades, several algorithms have been proposed to solve DCOPs, and they can be broadly classified into two classes: exact and non-exact. The former always provide an optimal solution of a given DCOP. However, solving DCOPs optimally is NP-hard [Modi *et al.*, 2005], thus scalability becomes an issue as the problem size grows. In contrast, non-exact algorithms compromise some solution quality for scalability. Among the non-exact algorithms, DSA [Zhang *et al.*, 2005], DSAN [Arshad and Silaghi, 2004], MGM & MGM2 [Maheswaran *et al.*, 2004a], Max-Sum [Farinelli *et al.*, 2008; Khan *et al.*, 2018a], Max-Sum_ADVP [Zivan and Peled, 2012], DSA-SDP [Zivan *et al.*, 2014], GDBA [Okamoto *et al.*, 2016], PD-Gibbs [Thien *et al.*, 2019] and AED [Mahmud *et al.*, 2020] are commonplace.

In general, the classical DCOP framework assumes that all the variables that are used to model a problem are discrete. However, many real-world applications (e.g. target tracking sensor orientation [Fitzpatrick and Meetrens, 2003], sleep scheduling of wireless sensors [Hsin and Liu, 2004]) are best modelled with continuous variables. In order to address this, Stranders *et al.* 2009 proposed a framework that facilitates the use of continuous variables, later referred to as a Functional DCOP (F-DCOP) [Choudhury *et al.*, 2020]. In contrast to classical DCOP, F-DCOP assumes all the variables are continuous and models all the constraints in the form of functions of those variables (instead of tabular form in classical DCOP). Among the F-DCOP solvers, CMS [Stranders *et al.*, 2009], HCMS [Voice *et al.*, 2010], D-Bay [Fransman *et al.*, 2020], EF-DPOP & AF-DPOP [Hoang *et al.*, 2019] and PFD [Choudhury *et al.*, 2020] are well known.

Against this background, it is obvious that the classical DCOP and F-DCOP can only deal with problems having discrete and continuous valued variables, respectively. In this paper, we first combine both of them into a Mixed Integer Functional DCOP (MIF-DCOP) framework, that can deal with a problem regardless of its variable types and representation of the constraints. We then develop a new algorithm that we call Distributed Parallel Simulated Annealing (DPSA) that can be directly applied to DCOPs and F-DCOPs, and even more importantly to their generalized version MIF-DCOPs.

The DPSA algorithm is based on Simulated Annealing (SA) meta-heuristic which is motivated by a physical analogy of controlled temperature cooling (i.e. annealing) of a material [Kirkpatrick *et al.*, 1983]. One of the most important factors that influence the quality of the solution produced by SA is this temperature parameter, widely denoted as T. More precisely, SA starts with a high value of T and during the search process continuously cools it down to near zero. When T is high, SA only explores the search space with-

out exploiting. This makes its behaviour similar to a random search procedure. On the other hand, when T is near zero, SA tends to only exploit and thus the exploration capability demises. In such a scenario, SA emulates the behaviour of a greedy algorithm. In fact, SA most effectively balances between exploration and exploitation in some optimal temperature region that lies in between these two extremes. Several existing works also discuss a constant optimal temperature where SA performs the best [Connolly, 1990; Alrefaei and Andradóttir, 1999]. Unfortunately, the optimal temperature region varies from one type of problem to another and from one instance to another of the same type problem (with different constraint function, constraint density, number of agents, etc.).

In light of the above, we introduce a novel method where agents cooperatively try to learn this optimal temperature region using a Monte Carlo importance sampling method called Cross-Entropy sampling [Kroese *et al.*, 2011] (discussed in the background). Using the learned knowledge during this process, agents also cooperatively solve the given problem. This results in a significant improvement of solution quality compared to the state-of-the-art algorithms in both DCOP and F-DCOP settings (see Section 5 for details). Moreover, we apply and evaluate both DSAN (i.e. the only other SA based DCOP solver) and DPSA in MIF-DCOP settings and show that DPSA outperforms DSAN in this setting by a notable margin.

2 Background

We first formally define DCOPs and F-DCOPs which will be the basis for the MIF-DCOP. We then conclude this section with a brief review of the literature necessary for this work.

2.1 DCOP Framework

A DCOP is defined by a tuple $\langle X, D, F, A, \delta \rangle$ [Modi *et al.*, 2005]. A is a set of agents $\{a_1, a_2, ..., a_n\}$. X is a set of discrete variables $\{x_1, x_2, ..., x_m\}$, which are being controlled by the set of agents A. D is a set of discrete and finite variable domains $\{D_1, D_2, ..., D_m\}$, where each D_i is a set containing values which may be assigned to its associated variable x_i. F is a set of constraints $\{f_1, f_2, ..., f_l\}$, where $f_i \in F$ is a function of a subset of variables $x^i \subseteq X$ defining the relationship among the variables in x^i. Thus, the function $f_i : \times_{x_j \in x^i} D_j \rightarrow R$ denotes the cost for each possible assignment of the variables in x^i. $\delta : X \rightarrow A$ is a variable-to-agent mapping function which assigns the control of each variable $x_i \in X$ to an agent of A [Khan *et al.*, 2018b]. Each variable is controlled by a single agent. However, each agent can hold several variables.

Within the framework, the objective of a DCOP algorithm is to produce X^*; a complete assignment that minimizes[1] the aggregated cost of the constraints as shown in Equation 1.

$$X^* = \underset{X}{\operatorname{argmin}} \sum_{i=1}^{l} f_i(x^i) \tag{1}$$

[1]For a maximization problem, replace argmin with argmax.

Algorithm 1: Cross-Entropy Sampling

```
Initialize parameter vector θ, #S, G, α
while condition not met do
    X ← take #S samples from distribution G(θ)
    S ← Evaluate points in X on the objective
    X ← sort(X, S)
    θ_new ← calculate updated θ using X(1:G)
    θ ← (1 − α) ∗ θ + α ∗ θ_new
```

For ease of understanding, we assume that each agent controls one variable. Thus, the terms 'variable' and 'agent' are used interchangeably throughout this paper.

2.2 Functional DCOP Framework

F-DCOPs can be defined by a tuple $\langle X, D, F, A, \delta \rangle$, where A and δ have the same definition as in the DCOP framework. However, the set of variables, X and the set of domains, D, are defined as follows: X is a set of continuous variables $\{x_1, x_2, ..., x_m\}$ that are controlled by agents in A. D is a set of continuous domains $\{D_1, D_2, ..., D_m\}$, where each variable x_i can take any value between a range, $D_i = [LB_i, UB_i]$. A notable difference between F-DCOP and DCOP is found in the representation of the cost function F. In DCOPs, the cost functions are conventionally represented in tabular form, while in F-DCOPs, each constraint is represented in the form of a function [Hoang *et al.*, 2019].

2.3 Distributed Simulated Annealing

Distributed Simulated Annealing (DSAN) [Arshad and Silaghi, 2004] is the only existing Simulated Annealing (SA) based DCOP solver. DSAN is a local search algorithm that executes the following steps iteratively:

- Each agent a_i selects a random value d_j from domain D_i.
- Agent a_i then assigns the selected value to x_i with the probability $min(1, exp(\frac{\Delta}{t_i}))$ where, Δ is the local improvement if d_j is assigned and t_i is temperature at iteration i. Note that authors of DSAN suggest that the value of $t_i = \frac{Max\_Iteration}{i^2}$ or $t_i = \frac{1}{i^2}$. However, setting the value of the temperature parameter with such a fixed schedule does not take into account their impact on the performance of the algorithm.
- Finally, agents notify neighbouring agents if the value of a variable changes.

2.4 Anytime Local Search

Anytime Local Search (ALS) is a general framework that gives distributed iterative local search DCOP algorithms such as DSAN described above an anytime property. Specifically, ALS uses a BFS-tree to calculate the global cost (i.e. evaluate Equation 1) of the system's state during each iteration and keeps track of the best state visited by the algorithm. Hence, using this framework, agents can carry out the best decision that they explored during the iterative search process instead of the one that occurs at the termination of the algorithm (see [Zivan *et al.*, 2014] for more details).

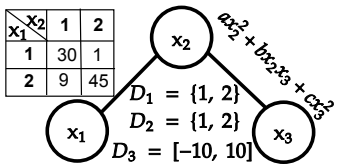

Figure 1: Example of a MIF-DCOPs.

2.5 Cross-Entropy Sampling

Cross-Entropy (CE) is a Monte Carlo method for importance sampling. CE has successfully been applied to importance sampling, rare-event simulation and optimization (discrete, continuous, and noisy problems) [Kroese *et al.*, 2011]. Algorithm 1 sketches an example that iteratively searches for the optimal value of the X within a search space. The algorithm starts with a probability distribution $\mathcal{G}(\theta)$ over the search space with parameter vector θ initialized to a certain value (that may be random). At each iteration, it takes $\#S$ (which is a parameter of the algorithm) samples from the probability distribution $\mathcal{G}(\theta)$ (line 3). After that, each sample point is evaluated on a problem dependent objective function. The top G among the $\#S$ sample points are used to calculate the new value of θ which is referred to as θ_{new} (lines $4-6$). Finally, θ_{new} is used to update θ (line 7). At the end of the learning process, most of the probability density of $\mathcal{G}(\theta)$ will be allocated near the optimal value of X.

3 Mixed Integer Functional DCOP Framework

We now formulate Mixed Integer Functional DCOP (MIF-DCOP) that combines both the classical DCOP and F-DCOP. This removes the requirement of all the variables being either continuous or discrete and constraint being represented in tabular or functional form. More formally, an MIF-DCOP is a tuple $\langle A, X, D, F, \delta \rangle$, where A, and δ are as defined in standard DCOPs. The key differences are as follows:

- X is a set of variables $\{x_1, x_2, ..., x_m\}$, where each $x_i \in X$ is either a discrete or a continuous variable.
- D is a set of domains $\{D_1, D_2, ..., D_m\}$. If a variable x_i is discrete, its domain D_i is the same as it is in the DCOP framework; otherwise, D_i is the same as it is in the F-DCOP model.
- F is a set of constraint functions $\{f_1, f_2, ..., f_l\}$. Each constraint function f_i can be modeled as follows: when the subset of the variables involved with f_i contains only discrete variables, it can be modeled either in tabular form or functional form. Otherwise it is modeled only in functional form. For example: Consider a binary constraint function $f(x_1, x_2)$ between two variables x_1 and x_2. Let $D_1 = [-1000, 1000]$ and $D_2 = \{red, green\}$. Example of $f(x_1, x_2)$ under MIF-DCOP is: $f(x_1, x_2) = I(x_2 = red) * f_{red}(x_1) + I(x_2 = green) * f_{green}(x_2)$ where $I(.)$ is an identity function.

It is worth highlighting that both DCOPs and F-DCOPs are special cases of MIF-DCOPs wherein either all the variables are discrete and constraints are in tabular form or all the variables are continuous and constraints are in functional form, respectively.

MIF-DCOPs are specifically useful when the variables in X represent decisions about a heterogeneous set of actions. Suppose, for instance, we want to model a target classification problem in a network of optical sensor agents. In the F-DCOP model of this problem [Stranders, 2010], an agent was only able to rotate their sensor to change its viewing direction. Now imagine that agents also can optically zoom in or zoom out to increase clarity or field of vision, respectively. The decision about rotation can be best modeled with a continuous variable (i.e. $rotation = [0°, 360°]$, as described in [Stranders, 2010]) and the decision about optical zoom is naturally modeled using discrete variables (e.g. $zoom = \{1x, 2x, 3x\}$). Other possible scenarios where MIF-DCOP can be applied are: mobile sensor applications where agents need to select both their location and the target they are covering and solving MIP problems in distributed settings. These problems, and many other similar problems where heterogeneous types of decision variables are needed, can easily be modelled with the newly proposed MIF-DCOPs. We provide a small example of MIF-DCOP in Figure 1.

4 The DPSA Algorithm

We will now describe the DPSA algorithm for solving MIF-DCOPs (Algorithm 2[2]). As discussed earlier, a big motivation behind DPSA is to learn the optimal temperature region for Simulated Annealing (SA). Thus, it is important that we formally define optimal temperature (Definition 1) and optimal temperature region (Definition 2).

Definition 1. *An Optimal Temperature given simulation length L* is a constant temperature at which the expected solution cost yielded by running SA for L iterations is the lowest of all the temperatures > 0.

Definition 2: *An Optimal Temperature Region (OTR) of length ϵ* is a continuous interval $[T_{min}, T_{max}]$ where $T_{max} - T_{min} \leq \epsilon$ and contains the optimal temperature. If we set T_{min} to near zero and T_{max} to a very large number, it will always be an OTR by the above definition; although not a useful one. The proposed DPSA algorithm tries to find an OTR with sufficiently small ϵ.

The DPSA algorithm consists of two main components: the parallel SA component and the learning component. The parallel SA component (lines $1-15$), is an extension of the existing DSAN algorithm that simulates K systems in parallel. Additionally, we introduce the Select_Next(·) function in which an agent uses different strategies to select a value from its domain depending on its variable type, Continuous or Discrete (line 13). This simple modification makes DPSA (also DSAN) applicable to DCOP, F-DCOP and MIF-DCOP. We also modify the existing ALS framework to make it compatible with parallel simulation.

The other significant component of DPSA is the iterative learning component sketched in the pseudo-code from line 16 to 36. It starts with a large temperature region and iteratively

[2]Select_Next(.), Scheduler(.) and Modified_ALS(·) are described in text.

Algorithm 2: The DPSA Algorithm

1 **Function** `Simulate(`*isLearning*`):`
2 **if** *isLearning is True* **then**
3 set $x_{i,k}\ \forall k \in \{1, ..., K\}$ to same random value
4 $L \leftarrow S_{len}$
5 **else**
6 set $x_{i,k}\ \forall k \in \{1, ..., K\}$ to value of x_i
7 $L \leftarrow Itr_{max} - R_{max} * S_{max} * S_{len}$
8 **for** $l = 1...L$ **do**
9 send value of $x_{i,k}\ \forall k \in \{1, ..., K\}$ to neighbouring agents
10 receive value of $x_{i,k}\ \forall k \in \{1, ..., K\}$ from neighbouring agents
11 Update x_i using Modified_ALS($\cdot$)
12 **for** $k = 1...K$ **do**
13 $v \leftarrow$`Select_Next(`D_i`)`
14 $t_k \leftarrow$`Scheduler(`l,k,*isLearning*`)`
15 Set $x_{i,k}$ to v with probability $min(1, \exp(\frac{\Delta_k}{t_k}))$
16 **Function** `Update_Parameter(`θ, E, T`):`
17 $Threshold \leftarrow$G-th best of set E
18 $SelectedSample \leftarrow \{t_k : e_k \leq Threshold\}$
19 Update θ using $SelectedSample$
20 **Function** `Main():`
21 Construct BFS Tree
22 Initialize parameters: Parameter Vector θ, $Itr_{max}, R_{max}, S_{max}, S_{len}, G, K$
23 **for** $R = 1...R_{max}$ *AND Conditions are met* **do**
24 $E \leftarrow \{e_1 = 0, e_2 = 0, ..., e_K = 0\}$
25 **if** *the agent is the root* **then**
26 $T \leftarrow \{t_1, t_2, ..., t_K\}$sampled from $\mathcal{G}(\theta)$
27 **else**
28 $T \leftarrow$ Receive T from the parent agent in the BFS-Tree
29 Send T to all the child agents in BFS-Tree
30 **for** $s = 1...S_{max}$ **do**
31 Synchronously start `Simulate` (True)
32 Wait for Modified_ALS($\cdot$) to terminate
33 **for** $e_k \in E$ **do**
34 $e_k \leftarrow e_k + \frac{bestCost_{S,k}}{S_{max}}$
35 `Update_Parameter` (θ, E, T)
36 Synchronously start `Simulate` (False)

tries to shrink it down to an OTR of a small length (ϵ). To obtain this, at each iteration, agents cooperatively perform actions (i.e. synchronously simulate parallel SA with different constant temperatures), collect feedback (i.e. the cost yields by different simulations) and use the feedback to update the current temperature region toward the goal region. The underlining algorithm that is used in the learning process is based on cross-entropy importance sampling. However, to make DPSA sample efficient, we present modifications that significantly reduce the number of iterations and parallel simulations needed.

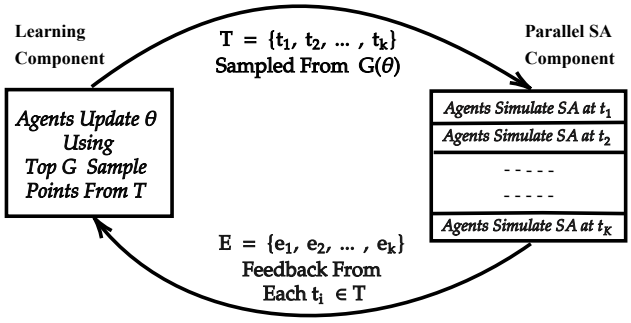

Figure 2: Overview of DPSA Algorithm

We structure the rest of our discussion in this section as follows: we first describe parallel SA and our modification of ALS. Then we discuss the learning component of DPSA and the techniques to make it sample efficient for DPSA. Finally, we analyze the complexity of DPSA.

In DPSA, the Simulate($\cdot$) function (lines $1-15$) runs SA on K copies of a system in parallel. This function is called in two different contexts: during learning to collect feedback (line 31) and after the learning process has ended (line 36). The main difference is that in the first context the function runs a short simulation in K different constant temperatures (one fixed temperature for each copy). In the second case, the function runs for significantly longer and all K systems run on the learned OTR with a fixed scheduler (discussed shortly). In addition, in the first case, all copies are initialized with the same random value from the domain. This is done because we want to identify the effect of different constant temperatures on the simulation step, and initializing them with different initial states would add more noise to the feedback. In the second case, we initialize with the best state found so far. Note that, to avoid confusion, we use x_i to refer to the actual decision variable and $x_{i,k}$ to refer to x_i on the k-th copy of the system. The parameter $isLearning$ (line 2) is used to represent the context in which the function was called. Depending on its value, variables of all K systems are initialized and the length of the simulation is set (lines $2-7$) (S_{len} is the simulation length during learning). After that, the main simulation starts and runs for L iterations.

At the start of each iteration, each agent a_i shares the current state of each system (i.e. the variable assignment of each system) with their neighbours (lines $9-10$). Each agent a_i then updates x_i and performs other operations related to Modified_ALS($\cdot$) (discussed shortly) (line 11). After that, for each of the K systems, agent a_i picks a value from its domain D_i (line 13) by calling the function $Select\_Next(\cdot)$. If D_i is discrete, it picks this value uniform randomly. If D_i is continuous, it picks it either using a Gaussian distribution $\mathcal{N}(x_{i,k}, \sigma)$ or a Uniform distribution $\mathcal{U}(LB_i, UB_i)$. Here, UB_i and LB_i is the bound of D_i. The difference is that Gaussian gives preference to a nearby value of the current assigned value, while uniform does not have any preference. Afterward, each agent selects the temperature for the current iteration i.e. t_k (line 14) by calling the function Scheduler($\cdot$). If it is called during the

learning context, it is always set to a constant. More precisely, it is set to the k-th value of T (from lines 26, 29). Otherwise, if the learned OTR is $[T_{min}, T_{max}]$; it can be used with a temperature scheduler for example linear scheduler. To calculate the temperature using linear scheduler we use Equation 2.

$$T_{min} + (T_{max} - T_{min})\frac{L - l}{L} \tag{2}$$

Finally, agents assign the value v to $x_{i,k}$ with the probability $min(1, \exp(\frac{\Delta_k}{t_k}))$ where Δ_k is the local gain (i.e. improvement of the aggregated cost of the local constraints if the assignment is changed) of the k-th system (line 15). If this gain is non-negative, it will always be changed (since $\exp(\frac{\Delta_k}{t_k}) \geq 1$). Otherwise, it will be accepted with a certain probability less than one.

We now describe our modification of ALS. This is used to collect feedback from the simulations during learning and to give DPSA its anytime property. We modify ALS in the following ways:

- Since DPSA simulated K systems in parallel, Modified_ALS(·) keeps track of the best state and the cost found by each system separately within the duration of a call of Simulate(·) function. This is used for the feedback.
- Modified_ALS(·) also keeps track of the best state and cost across all K systems and all the calls of the Simulate(·) function. Using this, agents assign values to their decision variables. This is used to give DPSA its anytime property.

The first part of the modification can easily be done by running each system at each call with its separate ALS. For the second part, we can have a meta-ALS that utilizes information of the ALSs in the first part to calculate the best state and cost across calls and systems.

We now discuss the learning component (lines $16 - 36$). Here, we start with a probability distribution $\mathcal{G}(\theta)$ over a large temperature region $[T_{min}, T_{max}]$ where θ is the parameter vector of the distribution. At the start of each iteration, the root agent samples K points (i.e. a set of constant temperatures T from this distribution (line 26)). Agents then propagate this information down the pseudo-tree (lines $28 - 29$). After that, agents synchronously call the Simulate(·) function for S_{max} times (line 31). At each call, agents simulate SA in the K sampled constant temperatures (i.e. set T) in parallel. Then using Modified_ALS(·), agents collect feedbacks i.e. cost of the best solution found by the simulation (line 32). Then agents take a mean over S_{max} feedback (lines $33-34$). This average should be an unbiased estimation of the expected solution quality i.e. the actual feedback given large S_{max}. Note that in the pseudo-code, we use $bestCost_{S,k}$ to refer to the best cost found by the k-th system in the S-th call. After all the feedback is collected, we use it to update the parameter vector using the Update_Parameter(·) function (line 35). The Update_Parameter(·) function takes the G best sample point (lines $17 - 18$) and uses them to update the parameter vector (line 19). In this way, agents iteratively learn the parameter vector θ.

The parameter vector θ and its update depends on the particular distribution $\mathcal{G}(\cdot)$ used. In this paper, we focus on two particular distributions namely Gaussian $\mathcal{N}(\cdot)$ and uniform $\mathcal{U}(\cdot)$. The parameter vector for $\mathcal{N}(\cdot)$ is $\theta = [\mu, \sigma]$ and consists of the mean and the standard deviation. The new parameter vector is calculated as:

$$\theta_{new} = [\mu(SelectedSample), \sigma(SelectedSample)] \tag{3}$$

The parameter vector for $\mathcal{U}(\cdot)$ is $\theta = [T_{min}, T_{max}]$ and consists of the current bound of temperature region. The new parameter vector is calculated as:

$$\theta_{new} = [min(SelectedSample), max(SelectedSample)] \tag{4}$$

Finally, θ is updated as shown in Equation 5 where α is the learning rate.

$$\theta = (1 - \alpha) * \theta + \alpha * \theta_{new} \tag{5}$$

Updating parameters in the way discussed above requires a considerable amount of iterations and samples. To reduce the number of iterations and samples required, we now discuss a few techniques that we used in the experimental section. First, when the number of parallel simulations is small (i.e. the value of K is small) taking a random sample in a large range is not efficient, and taking a stratified sample will cover a large range better. For example, when using $\mathcal{U}(\cdot)$, we may take samples at regular interval using Equation 6:

$$T_{min} + (T_{max} - T_{min}) * \frac{k - 1}{K - 1}; k \in \{1, 2, ..., K\} \tag{6}$$

Second, when S_{max} is small, the estimation of expected cost becomes noisy. To address this, when two sample points produce feedback within a bound of each other, we consider them equally good. We calculate this bound γ using Equation 7.

$$\gamma = \mathcal{S} * bestCost_* \tag{7}$$

Here, $\mathcal{S}$ stands for sensitivity and is an algorithm parameter and we use $bestCost_*$ to refer to the actual best cost found so far across all the calls of the Simulate(.) function. According to this, we may calculate the $Threshold$ in line 19 as follows: $Threshold \leftarrow$ G-th best of $E + \gamma$.

Finally, when R_{max} is small, setting learning rate to a larger value will speed up the learning process. However, if it is set too high, the algorithm might prematurely converge or skip the optimal temperature. Additionally, we can terminate before R_{max}, if all the sample points are within γ of each other. We now provide an example of the learning process:

Example 1 *Suppose, we have* $\alpha = 0.4, G = 3, K = 10, \theta = [0.1, 100]$ *and we use a uniform distribution. In the first round, the sampled points will be (when taken using the regular interval):*

$$T = [0.1, 11.1, 22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 100]$$

Let the feedback from each point be:

$$E = [50, 40, 30, 25, 32, 42, 57, 70, 95, 130]$$

The selected sample points (top $G = 3$ *points) will be:*

$$SelectedSample = [22.2, 33.3, 44.4]$$

Finally, the parameter update will be (min and max of $SelectedSample$):

$$\theta_{new} = [22.2, 44.4]$$

$$\theta = 0.6 * [0.1, 100] + 0.4 * [22.2, 44.4] = [8.9, 77.8]$$

This process will repeat until the termination conditions are met. We give a visual overview of the process in Figure 2.

After the learning process ends, agents call the Simulate(·) function for the final time at line 36. At this time, the simulation usually runs for longer on the learned optimal temperature region. This concludes our discussion on the learning component. It is worth noting that although we apply this learning component to learn parameter value for SA, it can also be applied to learn parameter(s) for other DCOP algorithms. In this way, it can be thought of as a generic distributed parameter learning algorithm for DCOPs.

In terms of complexity, the main cost is yielded by the Simulate(·) function. Each iteration of this function requires calculating the local gain Δ_k for K systems. The calculation of local gain requires $O(|N|)$ complexity where $|N|$ is the number of the neighbours. Hence, the computation complexity is $O(K|N|)$ (per iteration and per agent). In terms of communication complexity, K variable assignments are transferred at each iteration which gives it $O(K)$ complexity. Finally, agents have to save K local variable assignments, each of which requires $O(|N|)$ memory, meaning the total memory requirement will be $O(K|N|)$. It is worth mentioning that the memory requirement of Modified_ALS(·) is $O(K|H|)$ where $|H|$ is the height of the BFS tree. In Modified_ALS(·), while the number of messages remains the same as ALS size of each message increase by an factor of K.

5 Experimental Results

We start by evaluating DPSA against the state-of-the-art DCOP algorithms on 7 different benchmarks. We then test DPSA and DSAN against the state-of-the-art F-DCOP solvers. Finally, we present comparative solution quality produced by DPSA and DSAN on a MIF-DCOP setting.

For the former, we consider the following seven benchmarking DCOP algorithms: DSA-C (p = 0.8), MGM-2 (offer probability 0.5), Max-Sum ADVP, DSA-SDP (pA = 0.6, pB = 0.15, pC = 0.4, pD = 0.8), DSAN, GDBA and PD-Gibbs. For all the benchmarking algorithms, the parameter settings that yielded the best results are selected. We compare these algorithms on six Random DCOP settings and Weighted Graph Coloring Problems (WGCPs). For all settings, we use Erdős-Rényi topology (i.e. random graph) to generate the constraint graphs [Erdős and Rényi, 1960]. For random DCOPs, we vary the density p from 0.1 to 0.6 and the number of agents from 25 to 75. For WGCPs, we set $p = 0.05$ and number of agents to 120. We then take constraint costs uniformly from the range $[1, 100]$ and set domain size to 10. For all the benchmarks, we use the following parameters for DPSA $Itr_{max} = 2500, R_{max} = 12, S_{max} = 1, S_{lan} = 100, \alpha = 0.5, \mathcal{S} = .01$ and $K = 16$. When selecting values for these parameters note that increasing $Itr_{max}, R_{max}, S_{max}, S_{lan}$ and K will in general increase accuracy of the learning component in exchange for computational effort. A similar case is the learning rate α where decreasing it will increase accuracy but will slow down the learning process. Finally, for large values of S_{max}, sensitivity $\mathcal{S}$ will become less important. In all the settings of this section, DPSA uses $\mathcal{U}(\cdot)$ for CE and initializes parameter vector θ with a large temperature region $[10^{-4}, 10^{4}]$. In all of the settings described above, we run the benchmarking algorithms on 50 independently generated problems and 50 times on each problem for 500 ms. In order to conduct these experiments, we use a GCP-n2-highcpu-64 instance, a cloud computing service which is publicly accessible at cloud.google.com. Note that unless stated otherwise, all differences shown in this section are statistically significant for $p - value < 0.01$.

Algorithm	$\|A\|$ = 25		$\|A\|$ = 50		$\|A\|$ = 75	
	P = 0.1	P = 0.6	P = 0.1	P = 0.6	P = 0.1	P = 0.6
DSA-C	432	5725	2605	27163	7089	65519
DSA-SDP	325	5635	2365	27210	6711	65600
GDBA	386	5465	2475	26950	6867	65156
MGM-2	352	5756	2481	27421	6962	65988
PD-Gibbs	398	5875	2610	27350	7178	65650
MS_ADVP	400	5805	2550	27400	7058	66008
DSAN	408	5802	2639	27413	7224	66085
DPSA	**268**	**5358**	**2136**	**26240**	**6276**	**63998**

Table 1: Comparison of DPSA and the benchmarking algorithms on difference configuration of random DCOPs.

Figure 3 shows a comparison between DPSA and the benchmarking algorithms on the random DCOPs ($|A|$ = 75 and p=0.1) setting. While Table 1 presents how performances of these algorithms vary with the number of agents and density. When the density is low, the closest competitor to DPSA is DSA-SDP. Even though both of the algorithms keep on improving the solution until the end of the run, DPSA makes significant improvement when it starts running in the optimal temperature region after the learning process ends, and we see a big decline after 250 ms. From the results in Table 1, it can be observed that DPSA produces solutions that are $21\% - 6.7\%$ better than DSA-SDP depending on the number of agents. However, when the density is high ($p = 0.6$), GDBA is the closest competitor to DPSA. In dense settings, DPSA outperforms GDBA by $1.8\% - 1.9\%$. Other competing algorithms perform equal or worse than GDBA and DSA-SDP and produce even bigger performance difference with DPSA (up to 15% - 61% in sparse settings and 9.6% - 3.2% in dense settings). Also note that the optimal cost for ($|A|$ = 25, p = 0.1), which we generate using the well-known DPOP [Petcu and Faltings, 2005] algorithm, is 253, while DPSA produces 268 in the same setting.

Figure 4 shows a comparison between DPSA and the benchmarking algorithms on the WGCPs ($|A|$ = 120 and p=0.05) benchmark. We see a similar trend here as observed in the random DCOP settings. For the first 1200 iterations (up to 250 ms) i.e. during the learning stage, DPSA improves the solution with several small steps, and after that, it takes a big step toward a better solution when ran longer in the OTR. In this experiment, DPSA demonstrates its notable performance. Among the benchmarking algorithms, GDBA is

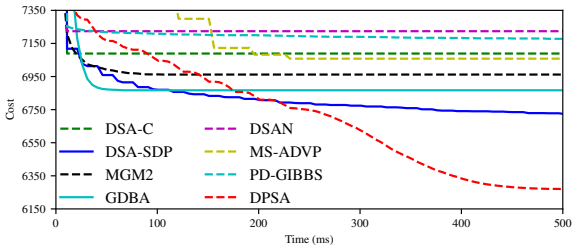

Figure 3: Comparison of DPSA and the benchmarking algorithms on random DCOPs ($|A|$ = 75, P = 0.1).

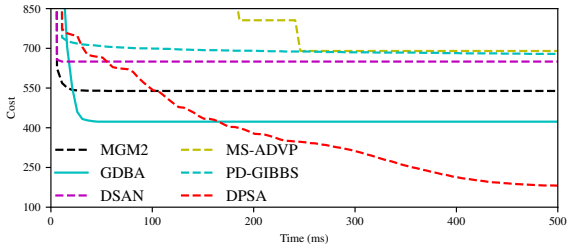

Figure 4: Comparison of DPSA and the benchmarking algorithms on weighted graph coloring problems ($|A|$ = 120, P = 0.05).

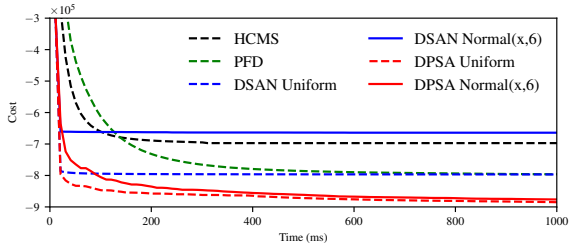

Figure 5: Comparison of DPSA and the benchmarking algorithms on binary quadratic F-DCOPs ($|A|$ = 50, P = 0.2).

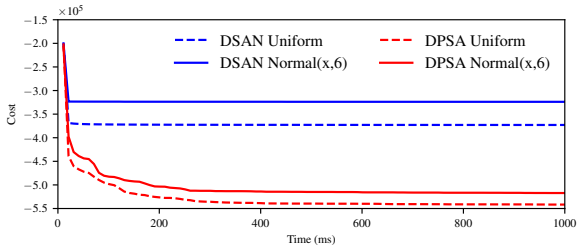

Figure 6: Comparison of DPSA and the benchmarking algorithms on binary quadratic MIF-DCOPs ($|A|$ = 50, P = 0.2).

the closest but is still outperformed by DPSA by 1.33 times. Among the other algorithms, DPSA finds solutions that are $3.65-1.95$ times better (3.65 times better than DSA-C). From the trend seen in Figures 3 and 4 and performance produced by DPSA compared to the current state-of-the-art DCOP algorithms signifies that DPSA applied in the optimal temperature region is extremely effective at solving DCOPs. Since both DPSA and DSAN apply the same principle; the big performance gain of DPSA in terms of solution quality can be credited to the fact that DPSA runs significantly longer near the optimal temperature.

We now compare DPSA and DSAN against current state-of-the-art F-DCOP solvers namely: PFD and HCMS on a large random graph with binary quadratic functions of the form $ax^2 + bxy + cy^2$. To generate random graphs, we use Erdős-Rényi topology with number of agents set to 50 and $p = 0.2$. We choose coefficients of the cost functions (a, b, c) randomly between $[-5, 5]$ and set the domains of each agent to $[-50, 50]$. We run all algorithms on 50 independently generated problems and 50 times on each problem for 1 second and use the same hardware setup as the previous settings. For PFD, we use the same configuration suggested in [Choudhury *et al.*, 2020]. For HCMS, we choose the number of discrete points to be 3. The discrete points are chosen randomly between the domain range. Finally, we use following parameters for DPSA $Itr_{max} = 3000, R_{max} = 12, S_{max} = 1, S_{lan} = 120, \alpha = 0.5, \mathcal{S} = 0.005$ and $K = 25$. To select neighbours in DPSA and DSAN, we use both uniform distribution and Normal distribution with $\sigma = 6$ over the domain.

Figure 5 shows a comparison between DPSA and the benchmarking algorithms on the binary quadratic F-DCOP ($|A|$ = 50, P = 0.2) benchmark. For this benchmark, uniform distribution for neighbour selection performs better than normal distribution both for DPSA and DSAN. DSAN (Uniform) produces similar solution quality as PFD. However, it has significantly improved anytime performance. On the other hand, DPSA produces solutions of significantly improved quality with the closest competitor PFD and DSAN being outperformed by 10.1%. This demonstrates that DPSA is also an effective F-DCOPs solver.

Finally, we test DPSA and DSAN in the MIF-DCOP setting. For this, we use the same set of problems as F-DCOPs except that we randomly make 50% of the variables discrete and set their domain to $\{-10, ..., 10\}$. Figure 6 shows anytime performance of DPSA and DSAN on the binary quadratic MIF-DCOP ($|A|$ = 50, P = 0.2) benchmark. We see a similar trend as we have seen in F-DCOPs benchmark. Here, DSAN converges fast to local optima and fails to make any further improvement. On the other hand, DPSA avoids local optima through maintaining a good balance between exploration and exploitation by operating in the optimal temperature region and produces a 46% better solution.

6 Conclusions

In this paper, we introduce MIF-DCOP framework that generalizes the well-known DCOP and F-DCOP models. We then propose a versatile algorithm called DPSA that can be applied to DCOPs, F-DCOPs and MIF-DCOPs. Finally, our empirical results depict that DPSA outperforms the state-of-the-art DCOP and F-DCOP algorithms by a significant margin and produces high quality solution compared to DSAN in MIF-DCOPs. In future, we intend to apply the learning component of DPSA to other DCOP algorithms such as DSA-SDP and Max-sum with damping to improve their solution quality.

Acknowledgements

This research is partially supported by the ICT Innovation Fund of Bangladesh Government and University Grants Commission (UGC) of Bangladesh.

References

[Alrefaei and Andradóttir, 1999] M. H. Alrefaei and S. Andradóttir. A simulated annealing algorithm with constant temperature for discrete stochastic optimization. *Management science*, 45:748–764, 1999.

[Arshad and Silaghi, 2004] M Arshad and M. C. Silaghi. Distributed simulated annealing. 2004.

[Choudhury *et al.*, 2020] M. Choudhury, M. Mahmud, and M.M. Khan. A particle swarm based algorithm for functional distributed constraint optimization problems. In *AAAI*, 2020.

[Connolly, 1990] D.T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46:93 – 100, 1990.

[Erdős and Rényi, 1960] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[Farinelli *et al.*, 2008] A. Farinelli, A. Rogers, A. Petcu, and N.R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *AAMAS*, 2008.

[Farinelli *et al.*, 2014] A. Farinelli, A. Rogers, and N.R. Jennings. Agent-based decentralised coordination for sensor networks using the max-sum algorithm. *Autonomous agents and multi-agent systems*, 28:337–380, 2014.

[Fioretto *et al.*, 2017] F. Fioretto, W. Yeoh, E. Pontelli, Y. Ma, and S.J. Ranade. A distributed constraint optimization (DCOP) approach to the economic dispatch with demand response. In *AAMAS*, 2017.

[Fitzpatrick and Meetrens, 2003] S. Fitzpatrick and L. Meetrens. Distributed sensor networks a multiagent perspective, chapter distributed coordination through anarchic optimization, 2003.

[Fransman *et al.*, 2020] J. Fransman, J. Sijs, H. Dol, E. Theunissen, and B. Schutter. Distributed bayesian: a continuous distributed constraint optimization problem solver. *arXiv preprint arXiv:2002.03252*, 2020.

[Hoang *et al.*, 2019] K.D. Hoang, W. Yeoh, M. Yokoo, and Z. Rabinovich. New algorithms for functional distributed constraint optimization problems. *arXiv preprint arXiv:1905.13275*, 2019.

[Hsin and Liu, 2004] C.n Hsin and M. Liu. Network coverage using low duty-cycled sensors: random & coordinated sleep algorithms. In *IPSN*, 2004.

[Khan *et al.*, 2018a] M.M. Khan, L. Tran-Thanh, and N.R. Jennings. A generic domain pruning technique for gdl-based dcop algorithms in cooperative multi-agent systems. In *AAMAS*, 2018.

[Khan *et al.*, 2018b] M.M. Khan, L. Tran-Thanh, W. Yeoh, and N.R. Jennings. A near-optimal node-to-agent mapping heuristic for GDL-based dcop algorithms in multi-agent systems. In *AAMAS*, 2018.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

[Kroese *et al.*, 2011] P.D. Kroese, T. Taimre, and Z.I. Botev. Handbook of monte carlo methods. 2011.

[Maheswaran *et al.*, 2004a] R.T. Maheswaran, J.P. Pearce, and M. Tambe. Distributed algorithms for dcop: A graphical-game-based approach. In *ISCA PDCS*, 2004.

[Maheswaran *et al.*, 2004b] R.T. Maheswaran, M. Tambe, E Bowring, J.P. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS*, 2004.

[Mahmud *et al.*, 2020] S. Mahmud, M. Choudhury, M.M. Khan, L. Tran-Thanh, and N.R. Jennings. AED: An anytime evolutionary dcop algorithm. In *AAMAS*, 2020.

[Modi *et al.*, 2005] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.

[Okamoto *et al.*, 2016] S. Okamoto, R. Zivan, and A. Nahon. Distributed breakout: Beyond satisfaction. In *IJCAI*, pages 447–453, 2016.

[Petcu and Faltings, 2005] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, 2005.

[Stranders *et al.*, 2009] R. Stranders, A. Farinelli, A. Rogers, and N.R. Jennings. Decentralised coordination of continuously valued control parameters using the max-sum algorithm. In *AAMAS*, 2009.

[Stranders, 2010] R. Stranders. Decentralised coordination of information gathering agents. 2010.

[Thien *et al.*, 2019] N. Thien, W. Yeoh, H. Lau, and R. Zivan. Distributed gibbs: A linear-space sampling-based dcop algorithm. *Journal of Artificial Intelligence Research*, 64:705–748, 2019.

[Voice *et al.*, 2010] T. Voice, R. Stranders, A. Rogers, and N.R. Jennings. A hybrid continuous max-sum algorithm for decentralised coordination. In *ECAI*, 2010.

[Yokoo *et al.*, 1998] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on knowledge and data engineering*, 10:673–685, 1998.

[Zhang *et al.*, 2005] W. Zhang, W. Wang, Z. Xing, and L. Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161:55–87, 2005.

[Zivan and Peled, 2012] R. Zivan and H. Peled. Max/min-sum distributed constraint optimization through value propagation on an alternating dag. In *AAMAS*, 2012.

[Zivan *et al.*, 2014] R. Zivan, S. Okamoto, and H. Peled. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence*, 212:1–26, 2014.