Distributed Gibbs: A Memory-Bounded Sampling-Based DCOP Algorithm[⋆]

Duc Thien Nguyen[†], William Yeoh[‡], and Hoong Chuin Lau[†]

[†]School of Information Systems
Singapore Management University
Singapore 178902
{dtnguyen.2011,hclau}@smu.edu.sg

[‡]Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
wyeoh@cs.nmsu.edu

Abstract. Researchers have used distributed constraint optimization problems (DCOPs) to model various multi-agent coordination and resource allocation problems. Very recently, Ottens *et al.* proposed a promising new approach to solve DCOPs that is based on confidence bounds via their Distributed UCT (DUCT) sampling-based algorithm. Unfortunately, its memory requirement per agent is exponential in the number of agents in the problem, which prohibits it from scaling up to large problems. Thus, in this paper, we introduce a new sampling-based DCOP algorithm called Distributed Gibbs, whose memory requirements per agent is linear in the number of agents in the problem. Additionally, we show empirically that our algorithm is able to find solutions that are better than DUCT; and computationally, our algorithm runs faster than DUCT as well as solve some large problems that DUCT failed to solve due to memory limitations.

1 Introduction

Distributed constraint optimization problems (DCOPs) are problems where agents need to coordinate their value assignments to maximize the sum of resulting constraint rewards [18, 21, 30]. Researchers have used them to model various multi-agent coordination and resource allocation problems such as the distributed scheduling of meetings [32], the distributed allocation of targets to sensors in a network [5, 33], the distributed allocation of resources in disaster evacuation scenarios [13], the distributed management of power distribution networks [12], the distributed generation of coalition structures [25] and the distributed coordination of logistics operations [14].

The field has matured considerably over the past decade as researchers continue to develop better and better algorithms. Most of these algorithms fall into

[⋆] A version of this paper appeared in AAMAS 2013 [19].

one of the following two classes of algorithms: (1) search-based algorithms like ADOPT [18] and its variants [29, 8], AFB [7] and MGM [16], where the agents enumerate through combinations of value assignments in a decentralized manner, and (2) inference-based algorithms like DPOP [21], max-sum [5] and Action GDL [26], where the agents use dynamic programming to propagate aggregated information to other agents.

More recently, Ottens *et al.* proposed a promising new approach to solve DCOPs that is based on confidence bounds [20]. They introduced a new sampling-based algorithm called Distributed UCT, which is an extension of UCB [1] and UCT [10]. While the algorithm is shown to outperform competing approximate and complete algorithms,its memory requirements per agent is exponential in the number of agents in the problem, which prohibits it from scaling up to large problems.

Thus, in this paper, we introduce a new sampling-based DCOP algorithm called Distributed Gibbs (D-Gibbs), which is a distributed extension of the Gibbs algorithm [6]. D-Gibbs is memory-bounded – its memory requirement per agent is linear in the number of agents in the problem. While the Gibbs algorithm was designed to approximate joint probability distributions in Markov random fields and solve maximum a posteriori (MAP) problems, we show how one can map such problems into DCOPs in order for Gibbs to operate directly on DCOPs. Our results show that D-Gibbs is able to find solutions that are better than DUCT faster than DUCT as well as solve some larger problems that DUCT failed to solve due to memory limitations.

2 Background: DCOP

A *distributed constraint optimization problem* (DCOP) [18, 17, 21] is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of finite domains, where D_i is the domain of variable x_i; $\mathcal{F}$ is a set of binary utility functions, where each utility function $F_{ij} : D_i \times D_j \mapsto \mathbb{N} \cup \{0, \infty\}$ specifies the utility of each combination of values of variables x_i and x_j; $\mathcal{A} = \{a_1, \dots, a_p\}$ is a set of agents and $\alpha : \mathcal{X} \to \mathcal{A}$ maps each variable to one agent. Although the general DCOP definition allows one agent to own multiple variables as well as the existence of n-ary constraints, we restrict our definition here for simplification purposes. One can transform a general DCOP to our DCOP using pre-processing techniques [31, 4, 2]. A solution is a value assignment for a subset of variables. Its utility is the evaluation of all utility functions on that solution. A solution is *complete* iff it is a value assignment for all variables. The goal is to find a utility-maximal complete solution.

A *constraint graph* visualizes a DCOP instance, where nodes in the graph correspond to variables in the DCOP and edges connect pairs of variables appearing in the same utility function. A *DFS pseudo-tree* arrangement has the same nodes and edges as the constraint graph and satisfies that (i) there is a subset of edges, called *tree edges*, that form a rooted tree and (ii) two variables in a utility function appear in the same branch of that tree. A DFS pseudo-tree

Algorithm 1: GIBBS($z_1, \ldots, z_n$)

```
1 for $i = 1$ to $n$ do
2     $z_i^0 \leftarrow$ INITIALIZE($z_i$)
3 end
4 for $t = 1$ to $T$ do
5     for $i = 1$ to $n$ do
6         $z_i^t \leftarrow$ SAMPLE($P(z_i \mid z_1^t, \ldots, z_{i-1}^t, z_{i+1}^{t-1}, \ldots, z_n^{t-1})$)
7     end
8 end
```

arrangement can be constructed using distributed DFS algorithms [9]. In this paper, we will use N_i to refer to the set of neighbors of variable x_i in the constraint graph, C_i to refer to the set of children of variable x_i in the pseudo-tree, and P_i to refer to the parent of variable x_i in the pseudo-tree.

3 Background: Algorithms

We now provide a brief overview of two relevant sampling-based algorithms – the centralized Gibbs algorithm and the Distributed UCT (DUCT) algorithm.

3.1 Gibbs

The Gibbs sampling algorithm [6] is a Markov chain Monte Carlo algorithm that can be used to approximate joint probability distributions. It generates a Markov chain of samples, each of which is correlated with previous samples. Suppose we have a joint probability distribution $P(z_1, z_2, \ldots, z_n)$ over n variables, which we would like to approximate. Algorithm 1 shows the pseudocode of the Gibbs algorithm, where each variable z_i^t represents the t-th sample of variable z_i. The algorithm first initializes z_i^0 to any arbitrary value of variable z_i (lines 1-3). Then, it iteratively samples z_i^t from the conditional probability distribution assuming that all the other $n-1$ variables take on their previously sampled values, respectively (lines 4-8). This process continues for a fixed number of iterations or until convergence, that is, the joint probability distribution approximated by the samples do not change. It is also common practice to ignore a number of samples at the beginning as it may not accurately represent the desired distribution. Once the joint probability distribution is found, one can easily identify that a complete solution with the maximum likelihood. This problem is called the *maximum a posteriori* (MAP) estimation problem, which is a common problem in applications such as image processing [3] and bioinformatics [28, 22].

The Gibbs sampling algorithm is desirable as its approximated joint probability distribution (formed using its samples) will converge to the true joint probability distribution given a sufficiently large number of samples for most problems. While Gibbs cannot be used to solve DCOPs directly, we will later show how one can slightly modify the problem such that Gibbs can be used to find optimal solutions given a sufficiently large number of samples.

3.2 Distributed UCT

The Upper Confidence Bound (UCB) [1] and UCB Applied to Trees (UCT) [10] algorithms are two Monte Carlo algorithms that have been successfully applied to find near optimal policies in large Markov Decision Processes (MDPs). The Distributed UCT (DUCT) algorithm [20] is a distributed version of UCT that can be used to find near-optimal cost-minimal complete DCOP solutions. We now provide a brief introduction to the algorithm and refer readers to the original article [20] for a more detailed treatment.

DUCT first constructs a pseudo-tree, after which each agents knows its parent, pseudo-parents, children and pseudo-children. Each agent x_i maintains the following for all possible contexts X and values $d \in D_i$:

- Its current value d_i.
- Its current context X_i, which is initialized to null. It is its assumption on the current values of its ancestors.
- Its cost y_i, which is initialized to ∞. It is the sum of the costs of all cost functions between itself and its ancestors given that they take on their respective values in its context and it takes on its current value.
- Its counter $\tau_i(X, d)$, which is initialized to 0. It is the number of times it has sampled value d under context X.
- Its counter $\tau_i(X)$, which is initialized to 0. It is the number of times it has received context X from its parent.
- Its cost $\hat{\mu}_i(X, d)$, which is initialized to ∞. It is the smallest cost found when it sampled d under context X so far up to the current iteration.
- Its cost $\hat{\mu}_i(X)$, which is initialized to ∞. It is the smallest cost found under context X so far up to the current iteration.

At the start, the root agent chooses its value and sends it down in a CONTEXT message to each of its children. When an agent receives a CONTEXT message, it too chooses its value, appends it to the context in the CONTEXT message, and sends the appended context down in a CONTEXT message to each of its children. Each agent x_i chooses its value d_i using:

$$d_i = \operatorname*{argmin}_{d \in D_i} B_i(d) \tag{1}$$

$$B_i(d) = f(\delta_i(d), \hat{\mu}_i(X_i, d), \tau_i(X_i, d), B_c) \tag{2}$$

$$\delta_i(d) = \sum_{\langle x_j, d_j \rangle \in X_i} F_{ij}(d, d_j) \tag{3}$$

where its bound $B_i(d)$ is initialized with a heuristic function f that balances exploration and exploitation. Additionally, each agent x_i increments the number of times it has chosen its current value d_i under its current context X_i using:

$$\tau_i(X_i, d_i) = \tau_i(X_i, d_i) + 1 \tag{4}$$

$$\tau_i(X_i) = \tau_i(X_i) + 1 \tag{5}$$

This process continues until leaf agents receive CONTEXT messages and choose their respective values. Then, each leaf agent calculates its cost and sends it up in a COST message to its parent. When an agent receives a COST message from each of its children, it too calculates its cost, which includes the costs received from its children, and sends it up to its parent. Each agent x_i calculates its costs y_i, $\hat{\mu}_i(X_i, d)$ and $\hat{\mu}_i(X_i)$ using:

$$y_i = \delta_i(d_i) + \sum_{x_c \in C_i} y_c \tag{6}$$

$$\hat{\mu}_i(X_i, d_i) = \min\{\hat{\mu}_i(X_i, d_i), y_i\} \tag{7}$$

$$\hat{\mu}_i(X_i) = \min\{\hat{\mu}_i(X_i), \hat{\mu}_i(X_i, d_i)\} \tag{8}$$

This process continues until the root agent receives a COST message from each of its children and calculates its own cost. Then, the root agent starts a new iteration, and the process continues until all the agents terminate. An agent x_i terminates if its parent has terminated and the following condition holds:

$$\max_{d \in D_i} \left\{ \hat{\mu}_i(X_i) - \left[\hat{\mu}_i(X_i, d) - \sqrt{\frac{\ln \frac{2}{\Delta}}{\tau_i(X_i, d_i)}} \right] \right\} \leq \epsilon \tag{9}$$

where Δ and ϵ are parameters of the algorithm.

4 Distributed Gibbs

While DUCT has been shown to be very promising, its memory requirement per agent is $O(\hat{D}^T)$, where $\hat{D} = \max_{x_i} D_i$ is the largest domain size over all agents and T is the depth of the pseudo-tree. Each agent needs to store a constant number of variables for all possible contexts and values,and the number of possible contexts is exponential in the number of ancestors. Therefore, this high memory requirement might prohibit the use of DUCT in large problems, especially if the agents have large domain sizes as well. Therefore, we now introduce the Distributed Gibbs algorithm, which is a distributed extension of the Gibbs algorithm adapted to solve DCOPs. Additionally, its memory requirement per agent is linear in the number of ancestors.

4.1 Mapping of MAP Estimation Problems to DCOPs

Recall that the Gibbs algorithm approximates a joint probability distribution over all the variables in a problem when only marginal distributions are available. Once the joint probability distribution is found, it finds the maximum a posteriori (MAP) solution. If we can map a DCOP where the goal is to find a complete solution with maximum utility, to a problem where the goal is to find a complete solution with the maximum likelihood, and that a solution with maximum utility is also a solution with maximum likelihood, then we can use Gibbs to solve DCOPs.

We now describe how to do so. Consider a maximum a posteriori (MAP) estimation problem on a pairwise Markov random field (MRF).[1] An MRF can be visualized by an undirected graph $\langle V, E \rangle$ and is formally defined by

- A set of random variables $\mathbf{X} = \{x_i \mid \forall i \in V\}$, where each random variable x_i can be assigned a value d_i from a finite domain D_i. Each random variable x_i is associated with node $i \in V$.
- A set of potential functions $\boldsymbol{\theta} = \{\theta_{ij}(x_i, x_j) \mid \forall (i,j) \in E\}$. Each potential function $\theta_{ij}(x_i, x_j)$ is associated with edge $(i,j) \in E$. Let the probability $P(x_i = d_i, x_j = d_j)$ be defined as $\exp(\theta_{ij}(x_i = d_i, x_j = d_j))$. For convenience, we will drop the values in the probabilities and use $P(x_i, x_j)$ to mean $P(x_i = d_i, x_j = d_j)$ from now on.

Therefore, a complete assignment $\boldsymbol{x}$ to all the random variables has the probability:

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{(i,j)\in E} \exp[\theta_{ij}(x_i, x_j)] = \frac{1}{Z} \exp\left[\sum_{(i,j)\in E} \theta_{ij}(x_i, x_j)\right] \tag{10}$$

where Z is the normalization constant. The objective of a MAP estimation problem is to find the most probable assignment to all the variables under $P(\boldsymbol{x})$, which is equivalent to finding a complete assignment $\boldsymbol{x}$ that maximizes the function:

$$F(\boldsymbol{x}) = \sum_{(i,j)\in E} \theta_{ij}(x_i, x_j) \tag{11}$$

Maximizing the function in Equation 11 is *also* the objective of a DCOP if each potential function θ_{ij} corresponds to a utility function F_{ij}. Therefore, if we use the Gibbs algorithm to solve a MAP estimation problem, then the complete solution found for the MAP estimation problem is also a solution to the corresponding DCOP.

4.2 Algorithm Description

We now describe the Distributed Gibbs algorithm. Algorithm 2 shows the pseudo-code, where each agent x_i maintains the following:

- Its values d_i and $\hat{d}_i$, which are both initialized to initial value **ValInit**(x_i). They are the agent's value in the current and previous iterations, respectively
- Its best value d_i^*, which is also initialized to initial value **ValInit**(x_i). It is the agent's value in the best solution found so far. Note that each agent maintains its own best value only and does not need to know the best values of other agents. The best solution $\boldsymbol{x}^* = (d_1^*, \ldots, d_n^*)$ can then be constructed upon termination.
- Its current context X_i, which is initialized with all the tuples of neighbors and their initial values. It is its assumption on the current values of its neighbors.

[1] We are describing pairwise MRFs so that the mapping to binary DCOPs is clearer.

Algorithm 2: Distributed Gibbs()

```
1  Create pseudo-tree
2  Each agent $x_i$ calls Initialize()
```

Procedure Initialize()

```
3  $d_i^* \leftarrow \hat{d}_i \leftarrow d_i \leftarrow$ ValInit($x_i$)
4  $X_i \leftarrow \{\langle x_j, \textbf{ValInit}(x_j)\rangle \mid x_j \in N_i\}$
5  $\Delta_i^* \leftarrow \Delta_i \leftarrow 0$
6  $t_i^* \leftarrow t_i \leftarrow 0$
7  if $x_i$ is root then
8      $t_i \leftarrow t_i + 1$
9      Sample()
10 end
```

- Its time index t_i, which is initialized to 0. It is the number of iterations it has sampled.
- Its time index t_i^*, which is initialized to 0. It indicates the most recent iteration that a better solution was found. The agents use it to know if they should update their respective best values.
- Its value Δ_i, which is initialized to 0. It is the difference in solution quality between the current solution and the best solution found in the previous iteration.
- Its value Δ_i^*, which is initialized to 0. It is the difference in solution quality between the best solution found in the current iteration and the best solution found so far up to the previous iteration.

The algorithm starts by constructing a pseudo-tree (line 1) and having each agent initialize their variables to their default values (lines 2-6). The root then starts by sampling, that is, choosing its value d_i based on the probability:

$$\begin{aligned} P(x_i \mid x_j \in \mathcal{X} \setminus \{x_i\}) &= P(x_i \mid x_j \in N_i) \\ &= \frac{1}{Z} \prod_{\langle x_j, d_j\rangle \in X_i} \exp[F_{ij}(d_i, d_j)] \\ &= \frac{1}{Z} \exp\left[\sum_{\langle x_j, d_j\rangle \in X_i} F_{ij}(d_i, d_j)\right] \end{aligned} \tag{12}$$

where Z is the normalization constant (lines 9 and 12). It then sends its value in a VALUE message to each of its neighbors (line 19).

When an agent receives a VALUE message, it updates the value of the sender in its context (line 20). If the message is from its parent, then it too samples and sends its value in a VALUE message to each of its neighbors (lines 32, 12 and 19). This process continues until all the leaf agents sample. Each leaf agent then sends a BACKTRACK message to its parent (line 34). When an agent receives a BACKTRACK message from each child (line 38), it too sends a BACK-

Procedure Sample()

```
11 $\hat{d}_i \leftarrow d_i$
12 $d_i \leftarrow$ Sample based on Equation 12
13 $\Delta_i \leftarrow \Delta_i + \sum_{\langle x_j,d_j\rangle \in X_i}[F_{ij}(d_i,d_j) - F_{ij}(\hat{d}_i,d_j)]$
14 if $\Delta_i > \Delta_i^*$ then
15     $\Delta_i^* \leftarrow \Delta_i$
16     $d_i^* \leftarrow d_i$
17     $t_i^* \leftarrow t_i$
18 end
19 Send VALUE $(x_i, d_i, \Delta_i, \Delta_i^*, t_i^*)$ to each $x_j \in N_i$
```

Procedure When Received VALUE($x_s, d_s, \Delta_s, \Delta_s^*, t_s^*$)

```
20 Update $\langle x_s, d'_s\rangle \in X_i$ with $(x_s, d_s)$
21 if $x_s = P_i$ then
22     Wait until received VALUE message from all pseudo-parents in this iteration
23     $t_i \leftarrow t_i + 1$
24     if $t_s^* = t_i$ then
25         $d_i^* \leftarrow d_i$
26     else if $t_s^* = t_i - 1$ and $t_s^* > t_i^*$ then
27         $d_i^* \leftarrow \hat{d}_i$
28     end
29     $\Delta_i \leftarrow \Delta_s$
30     $\Delta_i^* \leftarrow \Delta_s^*$
31     $t_i^* \leftarrow t_s^*$
32     Sample()
33     if $x_i$ is a leaf then
34         Send BACKTRACK $(x_i, \Delta_i, \Delta_i^*)$ to $P_i$
35     end
36 end
```

TRACK message to its parent (line 46). This process continues until the root agent receives a BACKTRACK message from each child, which concludes one iteration.

We now describe how the agents identify if they have found a better solution than the best one found thus far in a decentralized manner without having to know the values of every other agent in the problem. In order to do so, the agents use delta variables Δ_i and Δ_i^*. These variables are sent down the pseudo-tree in VALUE messages together with the current value of agents (line 19) and up the pseudo-tree in BACKTRACK messages (lines 34 and 46). When an agent receives a VALUE message from its parent, it updates its delta values to its parents' delta values prior to sampling (lines 29-30). After sampling, each agent calculates its local difference in solution quality $\sum_{\langle x_j,d_j\rangle \in X_i}[F_{ij}(d_i,d_j) - F_{ij}(\hat{d}_i,d_j)]$ and adds it to Δ_i (line 13). Thus, Δ_i can be seen as a sum of local differences from the root to the current agent as it is updated down the pseudo-tree. If this difference Δ_i is larger than the maximum difference Δ_i^*, which means

```
Procedure When Received BACKTRACK($x_s, \Delta_s, \Delta_s^*$)
37 Store $\Delta_s$ and $\Delta_s^*$
38 if Received BACKTRACK message from all children in this iteration then
39     $\Delta_i \leftarrow \big(\sum_{x_c \in C_i} \Delta_c\big) - \big(|C_i| - 1\big) \cdot \Delta_i$
40     $\Delta_{C_i}^* \leftarrow \big(\sum_{x_c \in C_i} \Delta_c^*\big) - \big(|C_i| - 1\big) \cdot \Delta_i^*$
41     if $\Delta_{C_i}^* > \Delta_i^*$ then
42         $\Delta_i^* \leftarrow \Delta_{C_i}^*$
43         $d_i^* \leftarrow d_i$
44         $t_i^* \leftarrow t_i$
45     end
46     Send BACKTRACK $(x_i, \Delta_i, \Delta_i^*)$ to $P_i$
47     if $x_i$ is root then
48         $\Delta_i \leftarrow \Delta_i - \Delta_i^*$
49         $\Delta_i^* \leftarrow 0$
50         $t_i \leftarrow t_i + 1$
51         Sample()
52     end
53 end
```

that the new solution is better than the best solution found thus far, then the agent updates the maximum difference Δ_i^* to Δ_i and its best value d_i^* to its current value d_i (lines 14-16).

After finding a better solution, the agent needs to inform other agents to update their respective best values to their current values since the best solution found thus far assumes that the other agents take on their respective current values. There are the following three types of agents that need to be informed:

- **Descendant agents:** The agent that has found a better solution updates its time index t_i^* to the current iteration (line 17) and sends this variable down to its children via VALUE messages (line 19). If an agent x_i receives a VALUE message from its parent with a time index t_s^* that equals the current iteration t_i, then it updates its best value d_i^* to its current value d_i (lines 24-25). It then updates its time index t_i^* to its parent's time index t_s^* and sends it down to its children via VALUE messages (lines 31, 32 and 19). This process continues until all descendant agents update their best values.
- **Ancestor agents:** The agent that has found a better solution sends its maximum difference Δ_i^* up to its parent via BACKTRACK messages (lines 34 and 46). In the simplest case where an agent x_i has only one child x_c, if the agent receives a BACKTRACK message with a maximum difference Δ_c^* larger than its own maximum difference, then it updates its best value d_i^* to its current value d_i (lines 40, 41 and 43). In the case where an agent has more than one child, then it compares the sum of the maximum differences over all children x_c subtracted by the overlaps (Δ_i^* was added an extra $|C_i|$ times) with its own maximum difference Δ_i^*. If the former is larger than the latter, then it updates its best value d_i^* to its current value d_i (lines 40, 41 and 43). It then updates its own maximum difference Δ_i^* (line 42) and sends it to its

parent via BACKTRACK messages (line 46). This process continues until all ancestor agents update their best values.

- **Sibling subtree agents:** Agents in sibling subtrees do not get VALUE or BACKTRACK messages from each other. Thus, an agent x_i cannot update its best value using the above two methods if another agent in its sibling subtree has found a better solution. However, in the next iteration, the common ancestor of these two agents will propagate its time index down to agent x_i via VALUE messages. If agent x_i receives a time index t_s^* that equals the previous iteration $t_i - 1$ and is larger than its own time index t_i^* (indicating that it hasn't found an even better solution in the current iteration), then it updates its best value d_i^* to its previous value $\hat{d}_i$ (lines 26-28). (It doesn't update its best value to its current value because the best solution was found in the previous iteration.) Thus, all agents in sibling subtrees also update their best values.

Therefore, when a better solution is found, all agents in the Distributed Gibbs algorithm update their best values by the end of the next iteration. The algorithm can either terminate after a given number of iterations or when no better solution is found for a given number of consecutive iterations. We later show that by choosing at least $\frac{1}{\alpha \cdot \epsilon}$ number of samples, the probability that the best solution found is in the top α-percentile is at least $1 - \epsilon$ (Theorem 2).[2]

4.3 Theoretical Properties

Like Gibbs, the Distributed Gibbs algorithm also samples the values sequentially and samples based on the same equation (Equation 12). The main difference is that Gibbs samples down a pseudo-chain (a pseudo-tree without sibling subtrees), while Distributed Gibbs exploits parallelism by sampling down a pseudo-tree. However, this difference only speeds up the sampling process and does not affect the correctness of the algorithm since agents in sibling subtrees are independent of each other. Thus, we will show several properties that hold for centralized Gibbs and, thus, also hold for Distributed Gibbs. Some of these properties are well-known (we label them "properties") and some are new properties (we label them "theorems") to the best of our knowledge. The proofs for the new properties are available in [19].

Property 1. Gibbs is guaranteed to converge.

Property 2. Upon convergence, the probability $P(\boldsymbol{x})$ of any solution $\boldsymbol{x}$ equals its approximated probability $P_{Gibbs}(\boldsymbol{x})$:

[2] One can slightly optimize the algorithm by having the agents (1) send their current values in BACKTRACK messages instead of VALUE messages to their parents; and (2) send smaller VALUE messages, which do not contain delta values and time indices, to all pseudo-children. We describe the unoptimized version here for ease of understanding.

$$P(\boldsymbol{x}) = P_{Gibbs}(\boldsymbol{x}) = \frac{\exp[F(\boldsymbol{x})]}{\sum_{\boldsymbol{x'} \in \mathcal{S}} \exp[F(\boldsymbol{x'})]}$$

where $\mathcal{S}$ is the set of all solutions sampled.

Property 3. The expected numbers of samples $\mathbf{N}_{Gibbs}$ to get optimal solution $\boldsymbol{x}^*$ is

$$E(\mathbf{N}_{Gibbs}) \leq \frac{1}{P_{Gibbs}(\boldsymbol{x}^*)} + L$$

where L is the number of samples needed before the estimated joint probability converges to the true joint probability.

The process of repeated sampling to get an optimal solution is equivalent to sampling Bernoulli trials with success probability $P_{Gibbs}(\boldsymbol{x}^*)$. Thus, the corresponding geometric variable for the number of samples needed to get an optimal solution for the first time has an expectation of $1/P_{Gibbs}(\boldsymbol{x}^*)$ [11]. In the following, we assume that $1/P_{Gibbs}(\boldsymbol{x}^*) >> L$ and we will thus ignore L.

Theorem 1. *The expected number of samples to find an optimal solution $\boldsymbol{x}^*$ with Gibbs is no greater than with a uniform sampling algorithm. In other words,*

$$P_{Gibbs}(\boldsymbol{x}^*) \geq P_{uniform}(\boldsymbol{x}^*)$$

Definition 1 *A set of top α-percentile solutions S_α is a set that contains solutions that are no worse than any solution in the supplementary set $D \setminus S$ and $\frac{|S_\alpha|}{|D|} = \alpha$.*

Theorem 2. *After $\boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}$ number of samples with Gibbs, the probability that the best solution found thus far $\boldsymbol{x_N}$ is in the top α-percentile is at least $1 - \epsilon$. In other words,*

$$P_{Gibbs}\left(\boldsymbol{x_N} \in S_\alpha \mid \boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}\right) \geq 1 - \epsilon$$

Corollary 1. *The quality of the solution found by Gibbs approaches optimal as the number of samples $\boldsymbol{N}$ approaches infinity. In other words,*

$$\lim_{\epsilon \to 0} P_{Gibbs}\left(\boldsymbol{x_N} \in S_\alpha \mid \boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}\right) = 1$$

While we have only described the above properties and theorems for problems with discrete values, we believe that they can be applied directly for DCOPs with continuous values [23] by changing the summations to integrations.

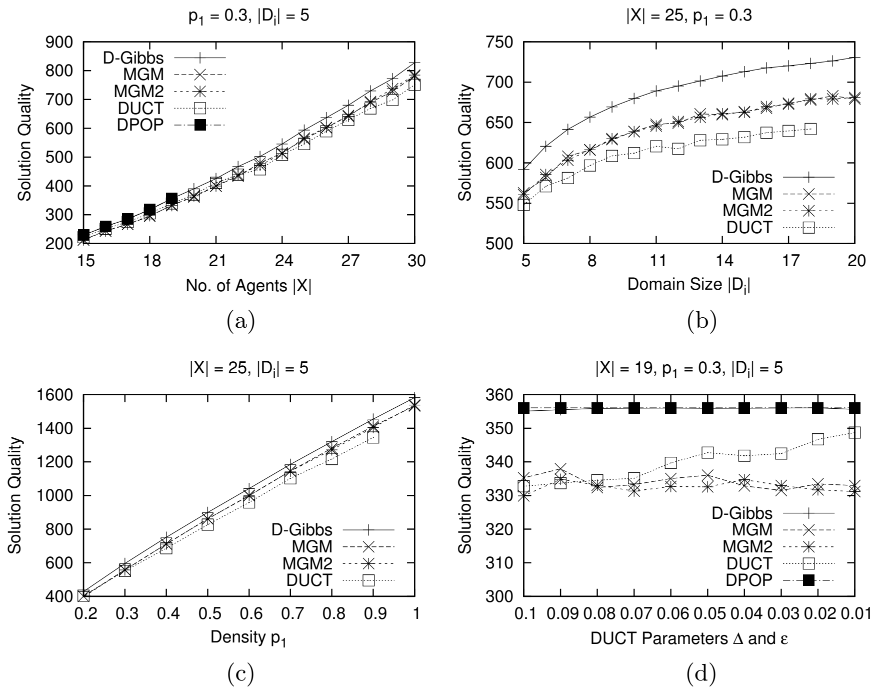

Fig. 1. Results for Graph Coloring Problems

4.4 Complexity Analysis

Each agent x_i needs to store a context X_i, which contains the agent-value pairs of all neighboring agents $x_j \in N_i$. Additionally agent x_i needs to store the delta values Δ_c and Δ_c^* for all children $x_c \in C_i$. Thus, the memory complexity of each agent is linear in the number of agents in the problem ($= O(|\mathcal{X}|)$).

Each agent x_i needs to send a VALUE message to each neighboring agent and a BACKTRACK message to its parent in each iteration, and each message contains a constant number of values (each VALUE message contains 5 values and each BACKTRACK message contains 3 values). Thus, the amount of information passed around in the network per iteration is polynomial in the number of agents in the problem ($= O(|\mathcal{X}|^2)$.

5 Experimental Results

We now compare Distributed Gibbs (D-Gibbs) to DPOP [21] (an optimal algorithm) and MGM [16], MGM2 [16] and DUCT [20] (sub-optimal algorithms). In terms of network load, that is, the amount of information passed around the network, DPOP sends an exponential amount of information in total ($= O(\exp(|\mathcal{X}|))$ while MGM, MGM2, DUCT and D-Gibbs send a polynomial amount of information in each iteration ($= O(|\mathcal{X}|^2)$.

To compare runtimes and solution qualities, we use publicly-available implementations of MGM, MGM2, DUCT and DPOP, which are all implemented on the FRODO framework [15]. We run our experiments on a 64 core Linux machine with 2GB of memory per run. We measure runtime using the simulated time metric [24] and evaluate the algorithms on graph coloring problems. For all problems, we set the DUCT parameters $\mathit{\Delta} = \epsilon = 0.05$, similar to the settings used in the original article [20] unless mentioned otherwise. We also let MGM, MGM2 and D-Gibbs run for as long as DUCT did for fair comparisons.[3] Each data point is averaged over 50 instances.

We used the random graph coloring problem generator provided in the FRODO framework [15] to generate our problems. We varied the size of the problem by increasing the number of agents $|\mathcal{X}|$ from 18 to 30, the graph density $p_1$from 0.2 to 1.0 and the domain size $|D_i|$ of each agent x_i from 5 to 20, and we chose the constraint utilities uniformly from the range (0, 10) at random if the neighboring agents have different values and 0 if they have the same value. Figure 1 shows our results, where we varied the number of agents $|\mathcal{X}|$ in Figure 1(a), the domain size $|D_i|$ in Figure 1(b), the density p_1 in Figure 1(c) and the DUCT parameters $\mathit{\Delta}$ and ϵ in Figure 1(d). DPOP ran out of memory for problems with 20 agents and above, and DUCT ran out of memory for problems with domain sizes 18 and 19 and for problems with a density of 1.

In all four figures, DPOP found better solutions (when it did not run out of memory) than D-Gibbs, which found better solutions than MGM, MGM2 and DUCT. The difference in solution quality increases as the number of agents, domain size and density increases.

Additionally, in Figure 1(d), as $\mathit{\Delta}$ and ϵ decreases, the runtimes of DUCT (and thus of all the other algorithms also since we let them run for as long as DUCT) increases since the tolerance for error decreases. However, the quality of its solutions improves as a result. Interestingly, the quality of solutions found by D-Gibbs, MGM and MGM2 remained relatively unchanged despite given more runtime, which means that they found their solutions very early on. Thus, D-Gibbs found close to optimal solutions faster (when $\mathit{\Delta} = \epsilon = 0.1$) than DUCT (when $\mathit{\Delta} = \epsilon = 0.01$).

6 Conclusions

Researchers have not investigated sampling-based approaches to solve DCOPs until very recently, where Ottens *et al.* introduced the Distributed UCT (DUCT) algorithm, which uses confidence-based bounds. However, one of its limitation is its memory requirement per agent, which is *exponential* in the number of agents in the problem. This large requirement prohibits it from scaling up to large problems. Examples include problems with domain sizes 19 and 20 or problems with a density of 1, which we showed experimentally. Therefore, in this paper, we introduce a new sampling-based algorithm called Distributed Gibbs (D-Gibbs),

[3] Exceptions are when DUCT failed to find a solution due to insufficient memory. For domain size $|D_i| = 19$ and 20 in Figure 1(b), we let the other algorithms run for as long as DUCT did for domain size $|D_i| = 18$, and for density $p_1 = 1$ in Figure 1(c), we let the other algorithms run for as long as DUCT did for density $p_1 = 0.9$.

whose memory requirement per agent is *linear* in the number of agents in the problem. It is a distributed extension of Gibbs, which was originally designed to approximate joint probability distributions in Markov random fields. We experimentally show that D-Gibbs finds better solutions compared to competing local search algorithms like MGM and MGM2 in addition to DUCT. Additionally, we also show how one can choose the number of samples based on the desired a priori approximation bound (using Theorem 2). While we have described D-Gibbs for (discrete-valued) DCOPs, we believe that it can easily be extended to solve continuous-valued DCOPs [23] as well. Thus, we would like to compare this approach with the Continuous-Valued Max-Sum [23, 27] in the future.

Acknowledgment

This research is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.
2. F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1–2):1–37, 2002.
3. J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society, Series B*, 48(3):259–279, 1986.
4. D. Burke and K. Brown. Efficiently handling complex local problems in distributed constraint optimisation. In *Proceedings of ECAI*, pages 701–702, 2006.
5. A. Farinelli, A. Rogers, A. Petcu, and N. Jennings. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of AAMAS*, pages 639–646, 2008.
6. S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.
7. A. Gershman, A. Meisels, and R. Zivan. Asynchronous Forward-Bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, 2009.
8. P. Gutierrez, P. Meseguer, and W. Yeoh. Generalizing ADOPT and BnB-ADOPT. In *Proceedings of IJCAI*, pages 554–559, 2011.
9. Y. Hamadi, C. Bessière, and J. Quinqueton. Distributed intelligent backtracking. In *Proceedings of ECAI*, pages 219–223, 1998.
10. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of ECML*, pages 282–293, 2006.
11. V. Kulkarni. *Modeling, Analysis, Design, and Control of Stochastic Systems.* Springer, 1999.
12. A. Kumar, B. Faltings, and A. Petcu. Distributed constraint optimization with structured resource constraints. In *Proceedings of AAMAS*, pages 923–930, 2009.
13. R. Lass, J. Kopena, E. Sultanik, D. Nguyen, C. Dugan, P. Modi, and W. Regli. Coordination of first responders under communication and resource constraints (Short Paper). In *Proceedings of AAMAS*, pages 1409–1413, 2008.

14. T. Léauté and B. Faltings. Coordinating logistics operations with privacy guarantees. In *Proceedings of IJCAI*, pages 2482–2487, 2011.
15. T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pages 160–164, 2009.
16. R. Maheswaran, J. Pearce, and M. Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of PDCS*, pages 432–439, 2004.
17. R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of AAMAS*, pages 438–445, 2004.
18. P. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.
19. D. T. Nguyen, W. Yeoh, and H. C. Lau. Distributed Gibbs: A memory-bounded sampling-based DCOP algorithm. In *Proceedings of AAMAS*, pages 167–174, 2013.
20. B. Ottens, C. Dimitrakakis, and B. Faltings. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of AAAI*, pages 528–534, 2012.
21. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of IJCAI*, pages 1413–1420, 2005.
22. D. Sontag, T. Meltzer, A. Globerson, T. Jaakkola, and Y. Weiss. Tightening LP relaxations for MAP using message passing. In *Proceedings of UAI*, pages 503–510, 2008.
23. R. Stranders, A. Farinelli, A. Rogers, and N. Jennings. Decentralised coordination of continuously valued control parameters using the Max-Sum algorithm. In *Proceedings of AAMAS*, pages 601–608, 2009.
24. E. Sultanik, R. Lass, and W. Regli. DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*, 2007.
25. S. Ueda, A. Iwasaki, and M. Yokoo. Coalition structure generation based on distributed constraint optimization. In *Proceedings of AAAI*, pages 197–203, 2010.
26. M. Vinyals, J. Rodríguez-Aguilar, and J. Cerquides. Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law. *Autonomous Agents and Multi-Agent Systems*, 22(3):439–464, 2011.
27. T. Voice, R. Stranders, A. Rogers, and N. Jennings. A hybrid continuous max-sum algorithm for decentralised coordination. In *Proceedings of ECAI*, pages 61–66, 2010.
28. C. Yanover, T. Meltzer, and Y. Weiss. Linear programming relaxations and belief propagation – an empirical study. *Journal of Machine Learning Research*, 7:1887–1907, 2006.
29. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.
30. W. Yeoh and M. Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.
31. M. Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
32. R. Zivan. Anytime local search for distributed constraint optimization. In *Proceedings of AAAI*, pages 393–398, 2008.
33. R. Zivan, R. Glinton, and K. Sycara. Distributed constraint optimization for large teams of mobile sensing agents. In *Proceedings of IAT*, pages 347–354, 2009.