Logic and Constraint Logic Programming
for Distributed Constraint Optimization

Tiep Le, Enrico Pontelli, Tran Cao Son, William Yeoh

Department of Computer Science, New Mexico State University
(*e-mail:* {tile,epontell,tson,wyeoh}@cs.nmsu.edu)

Abstract

The field of *Distributed Constraint Optimization Problems (DCOPs)* has gained momentum, thanks to its suitability in capturing complex problems (e.g., multi-agent coordination and resource allocation problems) that are naturally distributed and cannot be realistically addressed in a centralized manner. The state of the art in solving DCOPs relies on the use of ad-hoc infrastructures and ad-hoc constraint solving procedures. This paper investigates an infrastructure for solving DCOPs that is completely built on logic programming technologies. In particular, the paper explores the use of a general constraint solver (a *constraint logic programming system* in this context) to handle the agent-level constraint solving. The preliminary experiments show that logic programming provides benefits over a state-of-the-art DCOP system, in terms of performance and scalability, opening the doors to the use of more advanced technology (e.g., search strategies and complex constraints) for solving DCOPs.

KEYWORDS: DCOP, CLP, Implementation

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) are descriptions of constraint optimization problems where variables and constraints are distributed among a group of agents, and where each agent can only interact with agents that share a common constraint (Modi et al. 2005; Petcu and Faltings 2005; Yeoh and Yokoo 2012). Researchers have realized the importance of DCOPs, as they naturally capture real-world scenarios, where a collective tries to achieve optimal decisions, but without the ability to collect all information about resources and limitations into a central solver. For example, DCOPs have been successfully used to model domains like resource management and scheduling (Maheswaran et al. 2004; Farinelli et al. 2008; Léauté and Faltings 2011), sensor networks (Fitzpatrick and Meertens 2003; Jain and Ranade 2009; Zhang et al. 2005; Zivan et al. 2009; Stranders et al. 2009), and smart grids (Kumar et al. 2009; Gupta et al. 2013).

The DCOP field has grown at a fast pace in recent years. Several popular implementations of DCOP solvers have been created (Léauté et al. 2009; Sultanik et al. 2007; Ezzahir et al. 2007). The majority of the existing DCOP algorithms can be placed in one of three classes. *Search-based* algorithms perform a distributed search over the space of solutions to determine the optimum (Modi et al. 2005; Gershman et al. 2009; Zhang et al. 2005; Yeoh et al. 2010). *Inference-based* algorithms, on the other hand, make use of techniques

from dynamic programming to propagate aggregate information among agents (Petcu and Faltings 2005; Farinelli et al. 2008; Vinyals et al. 2009); these two classes provide a different balance between memory requirements and number of messages exchanged. Another class of methods includes approximated algorithms that rely on *sampling* (Ottens et al. 2012; Nguyen et al. 2013) applied to the overall search space.

The driving objective of the investigation discussed in this paper is to understand the role that logic programming can play in solving DCOPs. In particular, existing popular DCOP solvers (e.g., the frequently used FRODO platform (Léauté et al. 2009)) are ad-hoc systems, with a relatively closed structure, and making use of ad-hoc dedicated solvers for constraint handling within each agent. Thus, a question we intend to address with this paper is whether the use of a general infrastructure for constraint solving within each agent of a DCOP would bring benefits compared to the ad-hoc solutions of the existing implementations. We propose a general infrastructure (based on distributed dynamic programming) for the communication among agents, guaranteeing completeness of the system. The platform enables the use of a generic logic programming solver (e.g., a Constraint Logic Programming system) to handle the local constraints within each agent; the generality of the platform will also allow the use of distinct logic programming paradigms within each agent (e.g., Answer Set Programming).

The paper discusses the overall logic programming infrastructure, along with the details of the modeling of each agent using constraint logic programming. We provide some preliminary experimental results, validating the viability and effectiveness of this research direction for DCOPs. The results also highlight the potential offered by logic programming to provide an implicit representation of hard constraints in DCOPs, enabling a more effective pruning of the search space and reducing memory requirements.

2 Background

In this section, we provide a brief review of basic concepts from DCOPs. We assume that the readers have familiarity with logic and constraint logic programming; in particular, we will refer to the syntax of the clpfd library of SICStus Prolog (Carlsson et al. 2012).

2.1 Distributed Constraint Optimization Problems (DCOPs)

A *DCOP* (Modi et al. 2005; Petcu and Faltings 2005; Yeoh and Yokoo 2012) is described by a tuple $\mathcal{P} = (X, D, F, A, \alpha)$ where: **(i)** $X = \{x_1, \ldots, x_n\}$ is a set of *variables*; **(ii)** $D = \{D_{x_1}, \ldots, D_{x_n}\}$ is a set of finite *domains*, where each D_{x_i} is the domain of variable x_i; **(iii)** $F = \{f_1, \ldots, f_m\}$ is a set of *utility functions* (a.k.a. *constraints*), where each $f_j : D_{x_{j1}} \times D_{x_{j2}} \times \ldots \times D_{x_{jk}} \mapsto \mathbb{N} \cup \{-\infty, 0\}$ specifies the utility of each combination of values of variables in its *scope* $scp(f_j) = \{x_{j1}, \ldots, x_{jk}\} \subseteq X$; **(iv)** $A = \{a_1, \ldots, a_p\}$ is a set of *agents*; and **(v)** $\alpha : X \to A$ maps each variable to an agent.

We assume the domains D_x to be finite intervals of integer numbers. A *substitution* θ of a DCOP $\mathcal{P}$ is a value assignment for the variables in X s.t. $\theta(x) \in D_x$ for each $x \in X$. Its utility is $ut_{\mathcal{P}}(\theta) = \sum_{i=1}^{m} f_i(scp(f_i)\theta)$, i.e., the evaluation of all utility functions on it. A solution θ is a substitution such that $ut_{\mathcal{P}}(\theta)$ is maximal, i.e., there is no other substitution σ such that $ut_{\mathcal{P}}(\theta) < ut_{\mathcal{P}}(\sigma)$. $Soln_{\mathcal{P}}$ denotes the set of solutions of $\mathcal{P}$.

Each DCOP $\mathcal{P}$ is associated with a *constraint graph*, denoted with $G_{\mathcal{P}} = (X, E_{\mathcal{P}})$,

x_i	x_j	utility
0	0	5
0	1	8
1	0	20
1	1	2

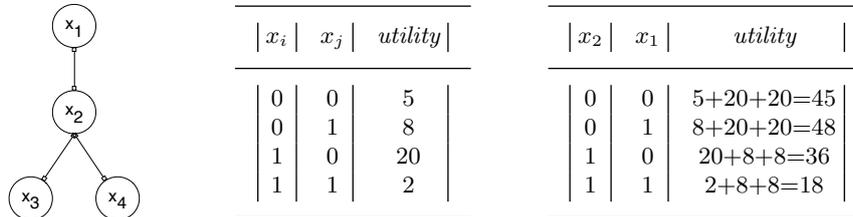| x_2	x_1	utility
0	0	5+20+20=45
0	1	8+20+20=48
1	0	20+8+8=36
1	1	2+8+8=18

Fig. 1. DCOP Example

where X is a set of nodes which correspond to DCOP variables, and $E_\mathcal{P}$ is a set of edges which connect pairs of variables in the scope of the same utility function.

2.2 Distributed Pseudo-tree Optimization Procedure (DPOP)

DPOP (Petcu and Faltings 2005) is one of the most popular complete algorithms for the distribution resolution of DCOPs; as discussed in several works, it has several nice properties (e.g., it requires only a linear number of messages), and it has been used as the foundations for several more advanced algorithms (Petcu et al. 2006; Petcu and Faltings 2007; Petcu et al. 2007).

The premise of DPOP is the generation of a *DFS-Pseudo-tree*—composed of a subgraph of the constraint graph of a DCOP. The pseudo-tree has a node for each agent in the DCOP; edges meet the following conditions: **(a)** If an edge (a_1, a_2) is present in the pseudo-tree, then there are two variables x_1, x_2 s.t. $\alpha(x_1) = a_1$, $\alpha(x_2) = a_2$, and $(x_1, x_2) \in E_\mathcal{P}$; **(b)** The set of edges describes a rooted tree; **(c)** For each pair of variables x_i, x_j s.t. $\alpha(x_i) \neq \alpha(x_j)$ and $(x_i, x_j) \in E_\mathcal{P}$, we have that $\alpha(x_i)$ and $\alpha(x_j)$ appear in the same branch of the pseudo-tree. $\alpha(x_i)$ and $\alpha(x_j)$ are also called the pseudo-parent and pseudo-child of each other.

Algorithms exist (e.g., (Hamadi et al. 1998)) to support the distributed construction of a DFS-Pseudo-tree. Given a DCOP $\mathcal{P}$, we will refer to a DFS-Pseudo-tree of $\mathcal{P}$ by $\mathcal{T}_\mathcal{P} = (A, ET_\mathcal{P})$. We will also denote with $a \mapsto_\mathcal{P} b$ if there exists a sequence of edges $(a_1, a_2), (a_2, a_3), \ldots, (a_{r-1}, a_r)$ in $ET_\mathcal{P}$ such that $a = a_1$ and $b = a_r$; in this case, we say that b is reachable from a in $\mathcal{T}_\mathcal{P}$. Given an agent a, we denote with $\mathcal{S}_\mathcal{P}(a)$ the set of agents in $\mathcal{T}_\mathcal{P}$ in the subtree rooted at a (including a itself).

The DPOP algorithm operates in two phases:

- **UTIL Propagation:** During this phase, messages flow bottom-up in the tree, from the leaves towards the root. Given a node N, the UTIL message sent by N summarizes the maximum utility achievable within the subtree rooted at N for each combination of values of variables belonging to the separator set (Dechter 2003) of N. The agent does so by summing the utilities in the UTIL messages received from its children agents, and then projecting out its own variables by optimizing over them.

- **VALUE Propagation:** During this phase, messages flow top-down in the tree. Node N determines an assignment to its own variables that produces the maximum utility based on the assignments given by the ancestor nodes; this assignment is then propagated as VALUE messages to the children.

Let us consider a DCOP with $X = \{x_1, x_2, x_3, x_4\}$, each with $D_{x_i} = \{0, 1\}$ and with binary constraints described by the graph (and pseudo-tree) and utility table (assuming

$i > j$) in Fig. 1 (left and middle). For simplicity, we assume a single variable per agent. Node x_2 will receive two UTIL messages from its children; for example, the message from x_3 will indicate that the best utilities are 20 (for $x_2 = 0$) and 8 (for $x_2 = 1$). In turn, x_2 will compose the UTIL messages with its own constraint, to generate a new utility table, shown in Fig. 1 (right). This will lead to a UTIL meassage sent to x_1 indicating utilities of 45 for $x_1 = 0$ and 48 for $x_1 = 1$. In the VALUE phase, node x_1 will generate an assignment of $x_1 = 1$, which will be sent as a VALUE message to x_2; in turn, x_2 will trigger the assignment $x_2 = 0$ as a VALUE message to its children.

3 Logic-Programing-based DPOP (LP-DPOP)

In this section, we illustrate the *LP-DPOP* framework, designed to map DCOPs into logic programs that can be solved in a distributed manner using the DPOP algorithm.
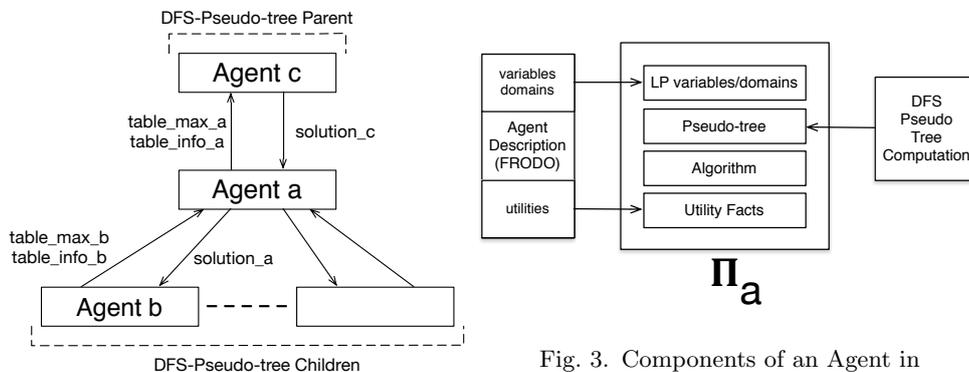

Fig. 2. Overall Communication Needs

Fig. 3. Components of an Agent in LP-DPOP$^{\text{facts}}$

3.1 Overall Structure

The overall structure of LP-DPOP is summarized in Fig. 2. Intuitively, each agent a of a DCOP $\mathcal{P}$ is mapped to a logic program Π_a. Agents exchange information according to the communication protocol of DPOP. These exchanges are represented by collections of facts that are communicated between agents. In particular,

- UTIL messages from agent b to agent a are encoded as facts *table_max_b(L)*, where L is a list of $[u, v_1, \ldots, v_k]$. Each one is a row of the UTIL message, where u is the maximum utility for the combination of values $v_1, \ldots, v_k$. It is also necessary to transmit an additional message describing the variables being communicated: *table_info_b([v(x_1, low_1, high_1), \ldots, v(x_k, low_k, high_k)])*. This message identifies the names of the variables being communicated and their respective domains. It should be mentioned that the UTIL message from b to a can contain variables belonging to some ancestors of a.
- VALUE messages from agent c to agent a are encoded as facts *solution_c(Var, Val)*, where *Var* is the name of a variable and *Val* is the value assigned to it.

3.2 LP-DPOP Execution Model

Computing DFS-PseudoTree: One can use existing off-the-shelf distributed algorithms to construct pseudo-trees. A commonly used algorithm is the distributed DFS

protocol (Hamadi et al. 1998), that creates a DFS tree with the max-degree heuristic as the variable-ordering heuristic. The max-degree heuristic favors variables with larger numbers of constraints to be higher up in the pseudo-tree.

Solving a DCOP: The actual agent a is implemented by a logic program Π_a. In the context of this paper, the logic program is a CLP program, whose entry point is a predicate called `agent`:

```
agent :- agent(ID),
         (\+is_leaf(ID) -> get_utils; true),
         (\+is_root(ID) -> compute_utils, send_utils, get_value; true),
         (\+is_leaf(ID) -> compute_value, send_value; compute_value).
```

The logic program implements the `compute_utils` and the `compute_value` predicates. They are described in the next section.

3.3 Modeling LP-DPOP as CLP

In this section, we illustrate the structure of the logic program that encodes each individual agent. We propose two alternative models. The first one follows the model illustrated in Fig 3: the input DCOP is described using the standardized format introduced by the FRODO DCOP platform (Léauté et al. 2009).

In the first model, referred to as LP-DPOP$^{\text{facts}}$, the FRODO model is literally translated into collections of logic programming facts. The second model, referred to as LP-DPOP$^{\text{rules}}$, follows the more "realistic" option of capturing the hard constraints present in the DCOP model explicitly as logical constraints, instead of forcing their mapping to explicit utility tables (as automatically done by FRODO).

3.3.1 LP-DPOP$^{\text{facts}}$

The logic program Π_a modeling an agent is composed of four primary modules, as illustrated in Fig. 3:

1. *Agent, Variables and Domains:* the core components of the agent variables and domains are encoded in Π_a by facts of the form:
   - A single fact $agent(a)$ describing the identity of the agent;
   - For each variable x_i with domain D_{x_i}, such that $\alpha(x_i) = a$ or $\alpha(x_j) = a$ for some variable x_j such that $(x_i, x_j) \in E_{\mathcal{P}}$: a fact $variable(x_i, min(D_{x_i}), max(D_{x_i}))$ and a fact $owner(\alpha(x_i), x_i)$.

2. *DFS-Pseudo-Tree:* the local position of a in the DFS-Pseudo-tree is described by:
   - facts of the form $child(b)$ where b is agent s.t. $(a, b) \in ET_{\mathcal{P}}$;
   - a fact $parent(c)$ where c is the (only) agent s.t. $(c, a) \in ET_{\mathcal{P}}$; and
   - a fact $ancestor(c)$ where c is any non-parent ancestor of a in the pseudo-tree, i.e., any agent c s.t. $(c, a) \notin ET_{\mathcal{P}}$ and $c \mapsto_{\mathcal{P}} a$.

3. *Utilities/Constraints:* the constraints are obtained as direct translation of the utility tables in the FRODO representation: for each constraint f_j, there is a fact of the form $constraint\_f_j(L)$, where L is a list containing lists $[f_j(v_1, \ldots v_r), v_1, \ldots, v_r]$ for each assignment $\{x_1/v_1, \ldots, x_r/v_r\}$ to the variables of $scp(f_j) = \{x_1, \ldots, x_r\}$ where $f_j(v_1, \ldots, v_r) \neq -\infty$. Each constraint is further described by the facts: *(i)* a fact $constraint(f_j)$, identifying the name of each constraint, *(ii)* a fact $scope(f_j, x_i)$

for each $x_i \in scp(f_j)$, identifying the variables contributing to the scope of the constraint, and *(iii)* facts of the form *constraint_agent*(f_j, a_r), identifying agents that has variables in the scope of the constraint.

4. *Resolution Engine:* a collection of rules that implement the `compute_utils` and `compute_value`—these are described below.

The core of the computation of the UTIL message is implemented within the `compute_utils` predicate. Intuitively, the construction of the UTIL message is mapped to a CLP problem. Its construction and resolution can be summarized as follows:

```
... define_variables(L,Low,High),
    define_constraints(L,Util),
    generate_utils(Low,High,UTILITIES), ...
```

The steps can be summarized as follows:

- The `define_variables` predicate is used to collect the variables that belong to the agent and its ancestors (returned in the list `Low` and `High`, respectively), and for each variable generates a corresponding CLP domain variable. The collecting variables phase is based on the *variable* facts (describing all variables owned by the agent) and the variables indicated in the *table_info_b* messages received from the children; these may contain variables that belong to pseudo-parents in the tree and unknown to the agent a. To enable interpretation of the CLP variables, two facts `low_vars`(Low) and `high_vars`(High) are created in this phase. In the latter phase, for each X_i in the collection of variables collected from the former phase calls X_i `in` $\ell..m$ where ℓ and m are the minimum and maximum value of X_i's domain which are either known to the agent or given in received the *table_info_b* message.

- The predicate `define_constraints` creates CLP constraints capturing the utilities the agent has to deal with—these include the utilities described by each *table_max_b* message received from a child b and the utilities f_j of the agent a s.t. $scp(f_j)$ does not contain any variables in $\bigcup_{(a,b) \in ET_{\mathcal{P}}} \{x \in X \mid \alpha(x) = b\}$. For each utility f_i of these utilities (described by a list of lists), the predicate `define_constraints` introduces a constraint of the form:

    ```
    table([[Ui, X1, .., Xr]] , L, [order(id3), consistency(domain)])
    ```
  where:
    - $X1, \ldots, Xr$ are the CLP variables which were created by `define_variables` and correspond to the scope of this utility.
    - L is the list of lists given in *constraint_f_i(L)*;
    - Ui is a new variable introduced for each utility f_i.

  The final step of the `define_constraints` is to introduce the additional CLP constraint $Util \#= U_1 + U_2 + \ldots + U_s$ where U_i are the variables introduced in the `table` constraints and $Util$ is a brand new variable.

- The `generate_utils` predicate has the following general structure:
    ```
    generate_utils(Lo, Hi, UTILITIES) :-
     findall([Util|Hi], (labeling([],Hi),find_max_util(Lo,Hi,Util)),UTILITIES).
    find_max_util(Lo, Hi, Util) :-
     maximize(labeling([ff],Lo), Util), assert(agent_a_table_max(Lo,Hi)).
    ```

The core of the computation of the VALUE message takes advantage of the fact that the combination of variables producing the maximum values are asserted as

`agent_a_table_max` facts during the UTILs phase, enabling a simple lookup to compute the solution. This can be summarized as follows:

```
... high_vars(H),
findall(Value,(member(Name,H),solution(Name,Value)), Sols),
agent_a_table_max(Low,Sols),
low_vars(Lo), length(Lo,Len), I in 1..Len,
findall(solution(Name,Value),
        (indomain(I), nth1(I,Lo,Name), nth1(I,Low,Value)), VALUES), ...
```

3.3.2 LP-DPOP^{rules}

An alternative encoding takes advantage of the fact that the utilities provided in the utility table of a FRODO encoding are the results of enumerating the solutions of *hard constraints*. A hard constraint captures a relation $f_j(x_1, \ldots, x_r) \oplus u$ where $\oplus$ is a relational operator, and u is an integer. This is typically captured in FRODO as a table, containing all tuples of values from $D_{x_1} \times \cdots \times D_{x_r}$ that satisfy the relation (with a utility value of 0), and the default utility value of $-\infty$ assigned to the remaining tuples.

This utility can be directly captured in CLP, thus avoiding the transition through the creation of an explicit table of solutions:

$$hard\_constraint\_f_j(X_1, \ldots, X_r) : -\widehat{f_j}(X_1, \ldots, X_r)\widehat{\oplus}u$$

where $\widehat{f_j}$ and $\widehat{\oplus}$ are the CLP operators corresponding to f_j and $\oplus$. For example, the smart grid problems used in the experimental section uses hard constraints encoded as

$$\texttt{hard\_constraint\_eq0}(X_{1,2}, X_{2,1}) :- X_{1,2} + X_{2,1}\# = 0$$

The resulting encoding of the UTIL value computation will modify the encoding of LP-DPOP^{facts} as shown below

```
constraint_f(L),
table([[U,X_1,...,X_r]],L,_)
```
$\implies$ `hard_constraint_f(X_1,...,X_r)`

3.4 Some Implementation Details

The current implementation of LP-DPOP makes use of the Linda (Carriero et al. 1994) infrastructure of SICStus Prolog (Carlsson et al. 2012) to handle all the communication.

Independent agents can be launched on different machines and connect to a Linda server started on a dedicated host. Each agent has a main clause of the type

`run_agent :- prolog_flag(argv, [Host,Port]),linda_client(Host:Port),agent.`

The operations of sending a UTIL message from b to the parent a is simply realized by a code fragment of the type

`send_util(Vars,Utils,To):- out(msg_to(To),[table_info_b(Vars),table_max_b(Utils)]).`

The corresponding reception of UTIL message by a will use a predicate of the form

`get_util(Vars,Utils,Me):- in(msg_to(Me), [table_info_b(Vars),table_max_b(Utils)]).`

The communication of VALUE messages is analogous. `get_value` and `send_value` are simple wrappers of the predicates discussed above.

3.5 Some Theoretical Considerations

The soundness and completeness of the LP-DPOP system is a natural consequence of the soundness and completeness properties of the DPOP algorithm, along with the soundness

and completeness of the CLP(FD) solver of SICStus Prolog. Since LP-DPOP emulates the computation and communication operations of DPOP, each Π_a program is a correct and complete implementation of the corresponding agent a.

In the worst case, each agent in LP-DPOP, like DPOP, needs to compute, store, and send a utility for each combination of values of the variables in the separator set of the agent. Therefore, like DPOP, LP-DPOP also suffers from an exponential memory requirement, i.e., the memory requirement per agent is $O(maxDom^w)$, where $maxDom = \text{argmax}_i |D_i|$ and w is the induced width of the pseudo-tree.

4 Experimental Results

We compare two implementations of the LP-DPOP framework, LP-DPOP$^{\text{facts}}$ and LP-DPOP$^{\text{rules}}$ with a publicly-available implementation of DPOP, which is available on the FRODO framework (Léauté et al. 2009). All experiments are conducted on a Quadcore 3.4GHz machine with 16GB of memory. The runtime of the algorithms are measured using the simulated runtime metric (Sultanik et al. 2007). The timeout is set to 10 minutes. Two domains, randomized graphs and smart grids, were used in the experiments.

Randomized Graphs: A randomized graph generated using the model in (Erdös and Rényi 1959) with the input parameters n (number of nodes) and M (number of binary edges) will be used as the constraint graph of a DCOP instance $\mathcal{P}$.

Each instance $\mathcal{P} = (X, D, F, A, \alpha)$ is generated using five parameters: $|X|$, $|A|$, the domain size d of all variables, the constraint density p_1 (defined as the ratio between the number of binary edges M and the maximum number of binary edges among $|X|$ nodes), and the constraint tightness p_2 (defined as the ratio between the number of infeasible value combinations, that is, their utility equals $-\infty$, and the total number of value combinations).

We conduct experiments, where we vary one parameter in each experiment. The "default" value for each experiment is $|A| = 5$, $|X| = 15$, $d = 6$, $p_1 = 0.6$, and $p_2 = 0.6$. As the utility tables of instances of this domain are randomly generated, the programs for LP-DPOP$^{\text{rules}}$ and LP-DPOP$^{\text{facts}}$ are very similar. Thus, we only compare FRODO with LP-DPOP$^{\text{facts}}$. Table 1 shows the percentage of instances solved and the average simulated runtime (in ms) for the solved instances; each data point is an average over 50 randomly generated instances. If an algorithm fails to solve more than 85% of instances in a specific configuration, then we consider that it fails to solve problems with that configuration.

The results show that LP-DPOP$^{\text{facts}}$ is able to solve more problems and is faster than DPOP when the problem becomes more complex (i.e., increasing $|X|$, d, p_1, or p_2). The reason is that at a specific percentage of hard constraints (i.e., $p_2 = 0.6$), LP-DPOP$^{\text{facts}}$ is able to prune a significant portion of the search space. Unlike DPOP, LP-DPOP$^{\text{facts}}$ does not need to explicitly represent the rows in the UTIL table that are infeasible, resulting in lower memory usage and runtime needed to search through search space. The size of the search space pruned increases as the complexity of the instance grows, making the difference between the runtimes of LP-DPOP$^{\text{facts}}$ and DPOP significant.

Smart Grids: A *customer-driven microgrid* (CDMG), one possible instantiation of the smart grid problem, has recently been shown to subsume several classical power system sub-problems (e.g., load shedding, demand response, restoration) (Jain et al. 2012). In this domain, each agent represents a node with consumption, generation, and transmis-

Table 1. *Experimental Results on Random Graphs (%: Solved; Time: Runtime)*

$|X|$	DPOP		LP-DPOP$^{\text{facts}}$		d	DPOP		LP-DPOP$^{\text{facts}}$	
	%	Time	%	Time		%	Time	%	Time
5	100%	35	100%	30	4	100%	782	100%	74
10	100%	204	100%	264	6	90%	28,363	100%	539
15	86%	39,701	100%	1,008	8	14%	-	98%	22,441
20	0%	-	100%	1,263	10	0%	-	94%	85,017
25	0%	-	100%	723	12	0%	-	60%	-
30	0%	-	100%	255					
35	0%	-	100%	256					

p_1	DPOP		LP-DPOP$^{\text{facts}}$		p_2	DPOP		LP-DPOP$^{\text{facts}}$	
	%	Time	%	Time		%	Time	%	Time
0.3	100%	286	100%	2,629	0.4	86%	48,632	92%	155,089
0.4	100%	1,856	100%	2,038	0.5	94%	38,043	100%	23,219
0.5	100%	13,519	100%	938	0.6	90%	31,513	100%	844
0.6	94%	42,010	100%	706	0.7	90%	39,352	100%	84
0.7	56%	-	100%	203	0.8	92%	40,525	100%	61
0.8	20%	-	100%	176	0.9	96%	27,416	100%	60

sion preference, and a global cost function. Constraints include the power balance and no power loss principles, the generation and consumption limits, and the capacity of the power line between nodes. The objective is to minimize a global cost function. CDMG optimization problems are well-suited to be modeled with DCOPs due to their distributed nature. Moreover, as some of the constraints in CDMGs (e.g., the power balance principle) can be described in functional form, they can be exploited by LP-DPOP$^{\text{rules}}$. For this reason, both LP-DPOP$^{\text{facts}}$ and LP-DPOP$^{\text{rules}}$ were used in this domain.

We conduct experiments on a range of CDMG problem instances generated using the four network topologies following the IEEE standards and varying the domain of the variables.[1] Fig. 4(a) displays the topology of the IEEE 13 Bus network, where rectangles represent nodes/agents, filled circles represent variables, and links between variables represent constraints. The initial configuration of the CDMG and the precise equations used in the generation of the problems can be found in (Jain et al. 2012). The experimental results for the four largest standards, the 13, 34, 37, and 123 Bus Topology,[2] are shown in Fig. 4(b), 4(c), 4(d), and 4(e), respectively. We make the following observations:

- LP-DPOP$^{\text{rules}}$ is the best among the three systems both in terms of runtime and scalability in all experiments. LP-DPOP$^{\text{rules}}$'s memory requirement during its execution is significant smaller and increases at a much slower pace than other systems. This indicates that the rules used in expressing the constraints help the constraint solver to more effectively prune the search space resulting in a better performance.
- LP-DPOP$^{\text{facts}}$ is slower than DPOP in all experiments in this domain. It is because LP-DPOP$^{\text{facts}}$ often needs to backtrack while computing the UTIL message, and each backtracking step requires the look up of several related utility tables—some

---

[1] www.ewh.ieee.org/soc/pes/dsacom/

[2] In 123 Bus Topology's experiments, a multi-server version of LP-DPOP$^{\text{facts}}$ and LP-DPOP$^{\text{rules}}$ was used because of the limit on the number of concurrent streams supported by Linda and SICStus. FRODO cannot be run on multiple machines.

(a) IEEE Standard 13 Bus Topology

(b) 13 Bus Topology

(c) 34 Bus Topology
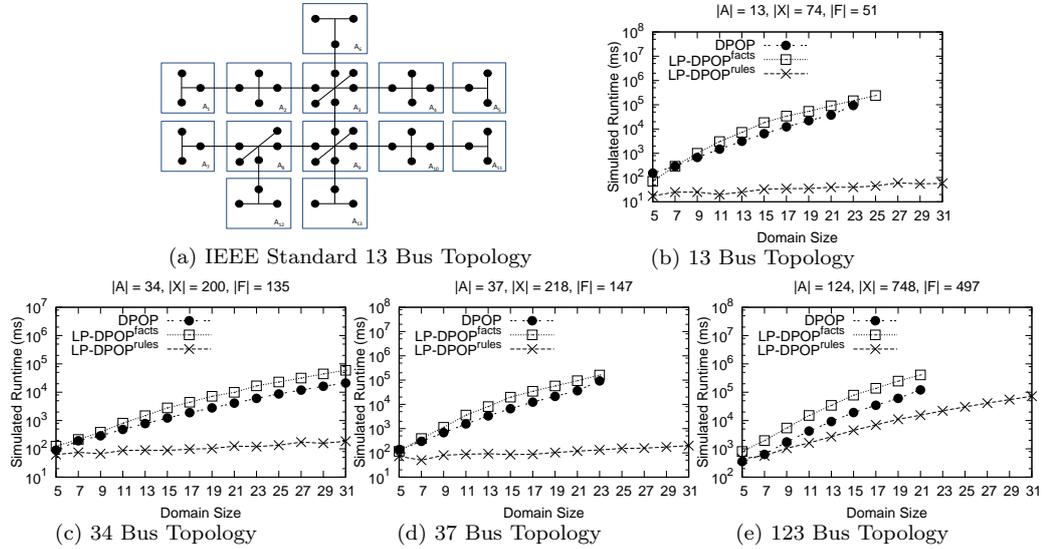
(d) 37 Bus Topology

(e) 123 Bus Topology

Fig. 4. Experiment Results on Smart Grids

tables can contain many tuples (e.g., one agent in the 13 Bus problem with domain size of 23 could have $3,543,173$ facts). We believe that this is the source of the weak performance of LP-DPOP$^{\text{facts}}$.

5 Conclusion and Future Work

In this paper, we presented a generic infrastructure built on logic programming to address problems in the area of DCOP. The use of a generic CLP solver to implement the individual agents proved to be a winning option, largely outperforming existing DCOP technology in terms of speed and scalability. The paper also makes the preliminary case for a different encoding of DCOPs w.r.t. existing technologies; the ability to explicitly model hard constraints provides agents with additional knowledge that can be used to prune the search space, further enhancing performance.

This is, in many regards, a preliminary effort that will be expanded in several directions. First, we believe that different types of DCOP problems may benefit from different types of local solvers within each agent; we currently explore the use of ASP as an alternative for the encoding the agents. The preliminary results are competitive and superior to those produced by DPOP. Classifying DCOP problems in such a way to enable the automated selection of what type of LP-based solver to use is an open research question to be addressed. The strong results observed in the use of implicit encodings of hard constraints also suggest the need of developing *DCOP description languages* that separate hard and soft constraints and do not require the explicit representation for all constraints.

On the other direction, we view this work as a feasibility study towards the development of distributed LP models (e.g., Distributed ASP). Paradigms like ASP are highly suitable to capture the description of individual agents operating in multi-agent environments; yet, ASP does not inherently provide the capability of handling a distributed ASP computation with properties analogous to those found in DCOP. We believe the models and infrastructure described in this paper could represent the first step in the direction of creating the foundations of DASP and other distributed logic programming models.

References

CARLSSON ET AL., M. 2012. SICStus Prolog User's Manual. Tech. rep., Swedish Institute of Computer Science.

CARRIERO, N., GELERNTER, D., MATTSON, T., AND SHERMAN, A. 1994. The Linda Alternative to Message Passing Systems. *Parallel Computing 20,* 4, 633–655.

DECHTER, R. 2003. *Constraint processing.* Elsevier Morgan Kaufmann.

ERDÖS, P. AND RÉNYI, A. 1959. On random graphs I. *Publicationes Mathematicae Debrecen 6,* 290.

EZZAHIR, R., BESSIERE, C., BELAISSAOUI, M., AND BOUYAKHF, E. H. 2007. DisChoco: A platform for distributed constraint programming. In *Proceedings of the Distributed Constraint Reasoning Workshop.* 16–27.

FARINELLI, A., ROGERS, A., PETCU, A., AND JENNINGS, N. 2008. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).* 639–646.

FITZPATRICK, S. AND MEERTENS, L. 2003. Distributed coordination through anarchic optimization. In *Distributed Sensor Networks: A Multiagent Perspective*, V. Lesser, C. Ortiz, and M. Tambe, Eds. Kluwer, 257–295.

GERSHMAN, A., MEISELS, A., AND ZIVAN, R. 2009. Asynchronous Forward-Bounding for distributed COPs. *Journal of Artificial Intelligence Research 34*, 61–88.

GUPTA, S., JAIN, P., YEOH, W., RANADE, S., AND PONTELLI, E. 2013. Solving customer-driven microgrid optimization problems as DCOPs. In *Proceedings of the Distributed Constraint Reasoning Workshop.* 45–59.

HAMADI, Y., BESSIÈRE, C., AND QUINQUETON, J. 1998. Distributed intelligent backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI).* 219–223.

JAIN, P. AND RANADE, S. 2009. Capacity discovery in customer-driven micro-grids. In *Proceedings of the North American Power Symposium (NAPS).* 1–6.

JAIN, P., RANADE, S., GUPTA, S., AND PONTELLI, E. 2012. Optimum operation of a customer-driven microgrid: A comprehensive approach. In *Proceedings of International Conference on Power Electronics, Drives and Energy Systems (PEDES).* 2012. 1–6.

KIEKINTVELD, C., YIN, Z., KUMAR, A., AND TAMBE, M. 2010. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).* 133–140.

KUMAR, A., FALTINGS, B., AND PETCU, A. 2009. Distributed constraint optimization with structured resource constraints. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).* 923–930.

LÉAUTÉ, T. AND FALTINGS, B. 2011. Coordinating logistics operations with privacy guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).* 2482–2487.

LÉAUTÉ, T., OTTENS, B., AND SZYMANEK, R. 2009. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the Distributed Constraint Reasoning Workshop.* 160–164.

MAHESWARAN, R., TAMBE, M., BOWRING, E., PEARCE, J., AND VARAKANTHAM, P. 2004. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).* 310–317.

MODI, P., SHEN, W.-M., TAMBE, M., AND YOKOO, M. 2005. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence 161,* 1–2, 149–180.

NGUYEN, D. T., YEOH, W., AND LAU, H. C. 2013. Distributed Gibbs: A memory-bounded

sampling-based DCOP algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 167–174.

OTTENS, B., DIMITRAKAKIS, C., AND FALTINGS, B. 2012. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 528–534.

PETCU, A. AND FALTINGS, B. 2005. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1413–1420.

PETCU, A. AND FALTINGS, B. 2007. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1452–1457.

PETCU, A., FALTINGS, B., AND MAILLER, R. 2007. PC-DPOP: A new partial centralization algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 167–172.

PETCU, A., FALTINGS, B., AND PARKES, D. C. 2006. MDPOP: Faithful distributed implementation of efficient social choice problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1397–1404.

PROSSER, P. 1996. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence 81,* 1-2, 81–109.

STRANDERS, R., FARINELLI, A., ROGERS, A., AND JENNINGS, N. 2009. Decentralised coordination of mobile sensors using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 299–304.

SULTANIK, E., LASS, R., AND REGLI, W. 2007. DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*.

VINYALS, M., RODRGUEZ-AGUILAR, J. A., AND CERQUIDES, J. 2009. Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1239–1240.

YEOH, W., FELNER, A., AND KOENIG, S. 2010. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research 38*, 85–133.

YEOH, W. AND YOKOO, M. 2012. Distributed problem solving. *AI Magazine 33,* 3, 53–65.

ZHANG, W., WANG, G., XING, Z., AND WITTENBERG, L. 2005. Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence 161,* 1–2, 55–87.

ZIVAN, R., YEDIDSION, H., OKAMOTO, S., GLINTON, R., AND SYCARA, K. 2014. Distributed constraint optimization for teams of mobile sensing agents. *Autonomous Agents and Multi-Agent Systems*, 1–42.