Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Solving Distributed Constraint Optimization Problems Using Logic Programming

Tiep Le, Tran Cao Son, Enrico Pontelli, William Yeoh
Department of Computer Science
New Mexico State University
{tile, tson, epontell, wyeoh}@cs.nmsu.edu

Abstract

This paper explores the use of *answer set programming* (ASP) in solving *distributed constraint optimization problems* (DCOPs). It makes the following contributions: (*i*) It shows how one can formulate DCOPs as logic programs; (*ii*) It introduces ASP-DPOP, the first DCOP algorithm that is based on logic programming; (*iii*) It experimentally shows that ASP-DPOP can be up to two orders of magnitude faster than DPOP (its imperative-programming counterpart) as well as solve some problems that DPOP fails to solve due to memory limitations; and (*iv*) It demonstrates the applicability of ASP in the wide array of multi-agent problems currently modeled as DCOPs.

Introduction

Distributed constraint optimization problems (DCOPs) are problems where agents need to coordinate their value assignments to maximize the sum of resulting constraint utilities (Modi et al. 2005; Petcu and Faltings 2005; Mailler and Lesser 2004; Yeoh and Yokoo 2012). Researchers have used them to model various multi-agent coordination and resource allocation problems (Maheswaran et al. 2004; Zivan, Glinton, and Sycara 2009; Zivan, Okamoto, and Peled 2014; Lass et al. 2008; Kumar, Faltings, and Petcu 2009; Ueda, Iwasaki, and Yokoo 2010; Léauté and Faltings 2011).

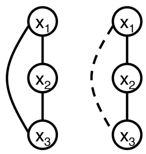
The field has matured considerably over the past decade, as researchers continue to develop better algorithms. Most of these algorithms fall into one of the following three classes: *(i)* search-based algorithms (Modi et al. 2005; Gershman, Meisels, and Zivan 2009; Zhang et al. 2005), where the agents enumerate combinations of value assignments in a decentralized manner; *(ii)* inference-based algorithms (Petcu and Faltings 2005; Farinelli et al. 2008), where the agents use dynamic programming to propagate aggregated information to other agents; and *(iii)* sampling-based algorithms (Ottens, Dimitrakakis, and Faltings 2012; Nguyen, Yeoh, and Lau 2013), where the agents sample the search space in a decentralized manner. The existing algorithms have been designed and developed almost exclusively using *imperative programming* techniques, where the algorithms define a control flow, that is, a sequence of commands to be executed. In addition, the local solver employed by each agent is an "ad-hoc" implementation. In this paper, we are interested in investigating the benefits of using *declarative programming* techniques to solve DCOPs, along with the use of a general constraint solver, used as a black box, as each agent's local constraint solver. Specifically, we propose an integration of DPOP (Petcu and Faltings 2005), a popular DCOP algorithm, with *answer set programming* (ASP) (Niemelä 1999; Marek and Truszczyński 1999) as the local constraint solver of each agent.

This paper contributes to both areas of DCOPs *and* ASP. For the DCOP community, we demonstrate that using ASP as the local constraint solver provides a number of benefits including the ability to capitalize on (*i*) the highly expressive ASP language to more concisely define input instances (e.g., by representing constraint utilities as implicit functions instead of explicitly enumerating them) and (*ii*) the highly optimized ASP solvers to exploit problem structure (e.g., propagating hard constraints to ensure consistency).

For the ASP community, while the proposed algorithm does not make the common contribution of extending the generality of the ASP language, it makes an *equally important* contribution of increasing the applicability of ASP to model and solve a wide array of multi-agent coordination and resource allocation problems currently modeled as DCOPs. Furthermore, it also demonstrates that general off-the-shelf ASP solvers, which are continuously honed and improved, can be coupled with distributed message passing protocols to outperform specialized imperative solvers, thereby validating the significance of the contributions from the ASP community. Therefore, in this paper, we make the first step of bridging the two areas of DCOPs and ASP in an effort towards deeper integrations of DCOP and ASP techniques that are mutually beneficial to both areas.

Background: DCOPs

A *distributed constraint optimization problem* (DCOP) (Modi et al. 2005; Petcu and Faltings 2005; Mailler and Lesser 2004; Yeoh and Yokoo 2012) is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of *variables*; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite *domains*, where D_i is the domain of variable x_i; $\mathcal{F} = \{f_1, \ldots, f_m\}$ is a set of *constraints*, where each k_i-ary constraint $f_i : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{k_i}} \mapsto \mathbb{N} \cup \{-\infty, 0\}$ specifies the utility of each combination of values of the variables in



(a) (b)

for $i < j$

x_i	x_j	Utilities
0	0	5
0	1	8
1	0	20
1	1	3

(c)

Figure 1: Example DCOP

x_1	x_2	Utilities
0	0	max(5+ 5, 8+8) = 16
0	1	max(5+20, 8+3) = 25
1	0	max(20+ 5, 3+8) = 25
1	1	max(20+20, 3+3) = 40

(a)

x_1	Utilities
0	max(5+16, 8+25) = 33
1	max(20+25, 3+40) = 45

(b)

Table 1: UTIL Phase Computations in DPOP

its *scope*, $scope(f_i) = \{x_{i_1}, \dots, x_{i_{k_i}}\}$; $\mathcal{A} = \{a_1, \dots, a_p\}$ is a set of *agents*; and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent. A *solution* is a value assignment for all variables and its corresponding utility is the evaluation of all utility functions on such solution. The goal is to find a utility-maximal solution.

A DCOP can be described by a *constraint graph*, where the nodes correspond to the variables in the DCOP, and the edges connect pairs of variables in the scope of the same utility function. A *DFS pseudo-tree* arrangement has the same nodes and edges as the constraint graph and: *(i)* there is a subset of edges, called *tree edges*, that form a rooted tree, and *(ii)* two variables in the scope of the same utility function appear in the same branch of the tree. The other edges are called *backedges*. Tree edges connect parent-child nodes, while backedges connect a node with its pseudo-parents and its pseudo-children. The *separator* of agent a_i is the set of variables owned by ancestor agents that are constrained with variables owned by agent a_i or by one of its descendant agents. A DFS pseudo-tree arrangement can be constructed using distributed DFS algorithms (Hamadi, Bessière, and Quinqueton 1998). Figure 1(a) shows a constraint graph of a DCOP with three agents, where each agent a_i controls variable x_i with domain $\{0, 1\}$. Figure 1(b) shows one possible pseudo-tree, where the dotted line is a *backedge*. Figure 1(c) shows the utility functions, assuming that all of the three constraints have the same function.

Background: DPOP

Distributed Pseudo-tree Optimization Procedure (DPOP) (Petcu and Faltings 2005) is a complete algorithm with the following three phases:

Pseudo-tree Generation Phase: DPOP calls existing distributed pseudo-tree construction methods (Hamadi, Bessière, and Quinqueton 1998) to build a pseudo-tree.

UTIL Propagation Phase: Each agent, starting from the leafs of the pseudo-tree, computes the optimal sum of utilities in its subtree for each value combination of variables in its separator. The agent does so by summing the utilities of its constraints with the variables in its separator and the utilities in the UTIL messages received from its child agents, and then projecting out its own variables by optimizing over them. In our DCOP example, agent a_3 computes the optimal utility for each value combination of variables x_1 and x_2 (see Table 1(a)), and sends the utilities to its parent agent a_2 in a UTIL message. For example, consider the first row of Table 1(a), where $x_1 = 0$ and $x_2 = 0$. The variable x_3 can be assigned a value of either 0 or 1, resulting in an aggregated utility value of (5+5=10) or (8+8=16), respectively. Then, the corresponding maximal value, which is 16, is selected to be sent to agent a_2. Agent a_2 then computes the optimal utility for each value of the variable x_1 (see Table 1(b)), and sends the utilities to its parent agent a_1. Finally, agent a_1 computes the optimal utility of the entire problem.

VALUE Propagation Phase: Each agent, starting from the root of the pseudo-tree, determines the optimal value for its variables. The root agent does so by choosing the values of its variables from its UTIL computations. In our DCOP example, agent a_1 determines that the value with the largest utility for its variable x_1 is 1, with a utility of 45, and then sends this information down to its child agent a_2 in a VALUE message. Upon receiving the VALUE message, agent a_2 determines that the value with the largest utility for its variable x_2, assuming that $x_1 = 1$, is 0, with a utility of 45, and then sends this information down to its child agent a_3. Finally, upon receiving the VALUE message, agent a_3 determines that the value with the largest utility for its variable x_3, assuming that $x_1 = 1$ and $x_2 = 0$, is 0, with a utility of 25.

Background: ASP

Let us provide some general background on *Answer Set Programming* (ASP). Consider a logic language $\mathcal{L} = \langle \mathcal{C}, \mathcal{P}, \mathcal{V} \rangle$, where $\mathcal{C}$ is a set of constants, $\mathcal{P}$ is a set of predicate symbols, and $\mathcal{V}$ is a set of variables. The notions of terms, atoms, and literals are the traditional ones.

An *answer set program* Π is a set of *rules* of the form

$$c \leftarrow a_1, \dots, a_m, not\ b_1, \dots, not\ b_n \tag{1}$$

where $m \geq 0$ and $n \geq 0$. Each a_i and b_i is a literal from $\mathcal{L}$, and each $not\ b_i$ is called a negation-as-failure literal (or naf-literal). c can be a literal or omitted. A program is a *positive program* if it does not contain naf-literals. A *non-ground rule* is a rule that contains variables; otherwise, it is called a *ground rule*. A rule with variables is simply used as a shorthand for the set of its ground instances from the language $\mathcal{L}$. If $n = m = 0$, then the rule is called a *fact*. If c is omitted, then the rule is called an *ASP constraint*.

A set of ground literals X is *consistent* if there is no atom a such that $\{a, \neg a\} \subseteq X$. A literal l is true (false) in a set of literals X if $l \in X$ ($l \notin X$). A set of ground literals X satisfies a ground rule of the form (1) if either of the following is true: *(i)* $c \in X$; *(ii)* some a_i is false in X; or *(iii)* some b_i is true in X. A solution of a program, called an *answer set* (Gelfond and Lifschitz 1990), is a consistent set of ground literals satisfying the following conditions:

- If Π is a *ground program* (a program whose rules are all ground) then its answer set S is defined by the following:

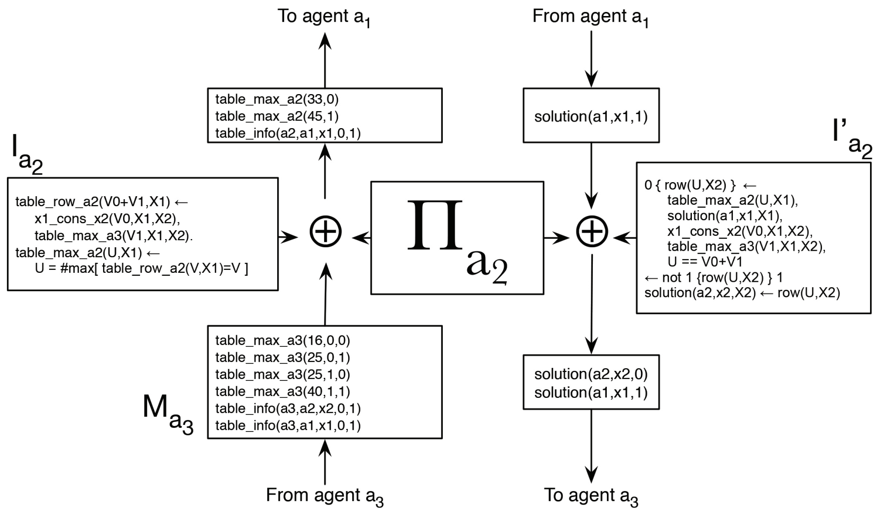


Figure 2: UTIL and VALUE Propagations of Agent a_2 in ASP-DPOP on our Example DCOP

- ◦ If Π does not contain any naf-literals, then S is an answer set of Π if it is a consistent and subset-minimal set of ground literals satisfying all rules in Π.
- ◦ If Π contains some naf-literals, then S is an answer set of Π if it is an answer set of the *program reduct* Π^S. Π^S is obtained from Π by deleting *(i)*) each rule that has a naf-literal *not* b in its body with $b \in S$, and *(ii)* all naf-literals in the bodies of the remaining rules.

- If Π is a *non-ground program*, that is, a program whose rules include non-ground rules, then S is an answer set of Π if it is an answer set of the program consisting of all ground instantiations of the rules in Π.

The ASP language includes also language-level extensions to facilitate the encoding of aggregates (min, max, sum, etc.), range specification of variables, and allowing choice of literals. In ASP, one solves a problem by encoding it as an ASP program whose answer sets correspond one-to-one to the solutions of the problem (Marek and Truszczyński 1999; Niemelä 1999). Answer sets of ASP programs can be computed using ASP solvers like CLASP (Gebser et al. 2007) and DLV (Citrigno et al. 1997).

Let us consider agent a_3 from the example DCOP. The utility tables of the constraints between agent a_3 and the other agents can be represented by the facts:

$$\begin{array}{ll} x_2\_cons\_x_3(5,0,0) & x_2\_cons\_x_3(8,0,1) \\ x_2\_cons\_x_3(20,1,0) & x_2\_cons\_x_3(3,1,1) \\ x_1\_cons\_x_3(5,0,0) & x_1\_cons\_x_3(8,0,1) \\ x_1\_cons\_x_3(20,1,0) & x_1\_cons\_x_3(3,1,1) \end{array}$$

The facts in the first two lines represent the constraint between agents a_2 and a_3. For example, the fact $x_2\_cons\_x_3(8,0,1)$ represents that a utility of 8 can be obtained if $x_2 = 0$ and $x_3 = 1$. The facts in the next two lines represent the constraint between agents a_1 and a_3. Agent a_3 can use these facts and the rule

$$table\_row\_a_3(U_1{+}U_2, Y, Z, X) \leftarrow x_2\_cons\_x_3(U_1, Y, X), \quad x_1\_cons\_x_3(U_2, Z, X).$$

to compute its UTIL table. Let us denote this program by Π_{a_3}. It is easy to see that Π_{a_3} has a unique answer containing the set of facts above and the following atoms:

$$\begin{array}{ll} table\_row\_a_3(10,0,0,0) & table\_row\_a_3(16,0,0,1) \\ table\_row\_a_3(25,0,1,0) & table\_row\_a_3(11,0,1,1) \\ table\_row\_a_3(25,1,0,0) & table\_row\_a_3(11,1,0,1) \\ table\_row\_a_3(40,1,1,0) & table\_row\_a_3(6,1,1,1) \end{array}$$

The first (resp. second) column lists four possible combinations when $x_3 = 0$ (resp. $x_3 = 1$). Intuitively, these are the atoms encoding the UTIL information of agent a_3. If we add to Π_{a_3} the rule

$$table\_max\_a_3(V, Y, Z) \leftarrow \quad V = \#max[table\_row\_a_3(U, Y, Z, \_) = U]$$

then the program has a unique answer set that contains the answer set of Π_{a_3} and the atoms $table\_max\_a_3(40, 1, 1)$, $table\_max\_a_3(25, 1, 0)$, $table\_max\_a_3(25, 0, 1)$, and $table\_max\_a_3(16, 0, 0)$. These atoms represent the information that agent a_3 will send to a_2 in a UTIL message.

ASP-DPOP

We now describe how to capture the structure of a DCOP in ASP and how to integrate the communication protocols of DPOP with the use of ASP as the internal controller/solver.

Specifying a DCOP using ASP

Let us consider a generic DCOP $\mathcal{P}$. We represent $\mathcal{P}$ using a set of logic programs $\{\Pi_{a_i} \mid a_i \in \mathcal{A}\}$. Each variable $x_j \in \mathcal{X}$ is described by a collection of rules $L(x_j)$ that include:

- A fact of the form $variable\_symbol(x_j)$, identifying the name of the variable.
- A fact of the form $value(x_j, d)$ for each $d \in D_j$, identifying the possible values of x_j. Alternatively, we can use additional facts of the form

$$begin(x_j, lower\_bound) \qquad end(x_j, upper\_bound)$$

to facilitate the description of domains. These facts denote the interval describing the domain of x_j. In such a case, the $value$ predicate is defined by the rule:

$$value(X, B..E) \leftarrow variable\_symbol(X), \\ begin(X, B), end(X, E)$$

Each utility function $f_i \in \mathcal{F}$ is encoded by a set of rules $L(f_i)$ of the form:

- A fact of the form $constraint(f_i)$, identifying the utility function.
- Facts of the form $scope(f_i, x_{i_j})$, identifying the variables in the scope of the constraint.
- Facts of the form $f_i(u, x, y)$, identifying the utility of a binary function f_i for a combination of admissible values of its variables. For higher arity utility functions, the rules can be expanded in a straightforward manner. It is also possible to envision the utility function modeled by rules that implicitly describe it.

For each agent a_i, the program Π_{a_i} consists of:

- A fact of the form $agent(a_i)$, identifying the name of the agent.
- A fact $neighbor(a_j)$ for each $a_j \in \mathcal{A}$, where $\alpha(x_k) = a_i$, $\alpha(x_t) = a_j$, and $\exists f_i \in \mathcal{F} | \{x_k, x_t\} \subseteq scope(f_i)$.
- A fact $owner(a_i, x_k)$ for each $x_k \in \mathcal{X}$, where $\alpha(x_k) = a_i$ or $\alpha(x_k) = a_j$ and a_j is a neighbor of a_i.
- A set of rules $L(f_j)$ for each utility function $f_j \in \mathcal{F}$ whose scope contains a variable owned by a_i; we refer to these utility functions as being *owned* by a_i;
- A set of rules $L(x_j)$ for each $x_j \in \mathcal{X}$ that is in the scope of a utility function that is owned by a_i.

Agent Controller in ASP-DCOP

The agent controller, denoted by C_{a_i}, consists of a set of rules for communication (sending, receiving, and interpreting messages) and a set of rules for generating an ASP program used for the computation of the utility tables as in Table 1 and the computation of the solution. We omit the detailed code of C_{a_i} due to space constraints. Instead, we will describe its functionality. For an agent a_i, C_{a_i} receives, from each child a_c of a_i, the UTIL messages consisting of facts of the form

- $table\_max\_a_c(u, v_1, \ldots, v_k)$, where u denotes the utility corresponding to the combination $v_1, \ldots, v_k$ of the set of variables $x_1, \ldots, x_k$; and
- $table\_info(a_c, a_p, x_p, lb_p, ub_p)$, where a_p is an ancestor of a_c who owns the variable x_p, and the lower and upper bounds lb_p and ub_p of that variable's domain.

Facts of the form $table\_info(a_c, a_p, x_p, lb_p, ub_p)$ encode the information necessary to understand the mapping between the values $v_1, \ldots, v_k$ in the $table\_max\_a_c$ facts and the corresponding variables (and their owners) of the problem.

C_{a_i} will use the agent's specification (Π_{a_i}) and the UTIL messages from its children to generate a set of rules that will be used to compute the UTIL message (e.g., Table 1(a) for agent a_3 and Table 1(b) for agent a_2) and solution extraction. In particular, the controller will generate rules for the following purposes:

- Define the predicate $table\_row\_a_i(U, X_1, \ldots, X_n)$, which specifies how the utility of a given combination value of variables is computed; e.g., for agent a_2

$$table\_row\_a_2(U + U_0, 0) \leftarrow \\ table\_max\_a_3(U, 0, X_2), x_1\_cons\_x_2(U_0, 0, X_2).$$

- Identify the maximal utility for a given value combination of its separator set; e.g., for agent a_2

$$table\_max\_a_2(U_{max}, 0) \leftarrow \\ U_{max} = \#max[table\_row\_a_2(U, 0) = U].$$

- Select the optimal row from the utility table. For example, the rule that defines the predicate row for agent a_2 in our running example is as follows:

$$\{row(U, X_2)\} \leftarrow solution(a_1, x_1, X_1), table\_max\_a_2(U, X_1), \\ table\_max\_a_3(U_0, X_1, X_2), \\ x_1\_cons\_x_2(U_1, X_1, X_2), U == U_0 + U_1. \\ \leftarrow not\ 1\ \{row(U, X_2)\}\ 1.$$

where facts of the form $solution(a_s, x_t, v_t)$ indicate that a_s, an ancestor of a_i, selects the value v_t for x_t.

This set of rules needs to be generated dynamically since it depends on the arity of the constraints owned by the agents. Also, it depends on whether a_i is the root, a leaf, or an inner node of the pseudo-tree. For later use, let us denote with I_{a_i} the set of rules defining the predicates $table\_row\_a_i$ and $table\_max\_a_i$, and with I'_{a_i} the set of rules defining the predicate row.

ASP-DPOP Description

Let us now describe ASP-DPOP, a complete ASP-based DCOP algorithm that emulates the computation and communication operations of DPOP. Like DPOP, there are three phases in the operation of ASP-DPOP.

At a high level, each agent a_i in the system is composed of two main components: the agent specification Π_{a_i} and its controller C_{a_i}. The two steps of propagation—generation of the table to be sent from an agent to its parent in the pseudo-tree (UTIL propagation) and propagation of variable assignments from an agent to its children in the pseudo-tree (VALUE propagation)—are achieved by computing solutions of two ASP programs.

Pseudo-tree Generation Phase: Like DPOP, ASP-DPOP calls existing distributed pseudo-tree construction algorithms to construct its pseudo-tree. The information about the parent, children, and pseudo-parents of each agent a_i are added to Π_{a_i} at the end of this phase.

UTIL Propagation Phase: C_{a_i} receives utilities as the set of facts M_{a_j} from the children a_j and generates the set of rules I_{a_i}. For example, Figure 2 shows I_{a_2} and M_{a_3} for agent a_2 of the DCOP in Figure 1. An answer set of the program $\Pi_{a_i} \cup I_{a_i} \cup \bigcup_{children\ a_j} M_{a_j}$ is computed using an ASP solver. Facts of the form $table\_max\_a_i(u, x_1, \ldots, x_k)$ and $table\_info(a_i, a_p, x_p, lb_p, ub_p)$ are extracted and sent to the parent a_p of a_i as M_{a_i}. This set is also kept for computing the solution of a_i.

VALUE Propagation Phase: The controller generates I'_{a_i} and computes the answer set of the program $\Pi_{a_i} \cup I'_{a_i}$ with the set of facts of the form $table\_max\_a_i(u, x_1, \ldots, x_k)$ and the set of facts of the form $solution(a_s, x_t, v_t)$. It then extracts the set of atoms of the form $solution(a_i, x_j, v_j)$ and sends them to its children.

In summary, the UTIL and VALUE propagations correspond to one ASP computation each (see Figure 2). We use CLASP (Gebser et al. 2007) with its companion grounder GRINGO, as our ASP solver, being the current state-of-the-art for ASP. As the solver does not provide any communication capabilities, we use the *Linda* facilities offered by SICStus© Prolog for communication. The controller C_{a_i} handles UTIL propagation and VALUE propagation using a Linda blackboard to exchange the facts as illustrated earlier.

Theoretical Properties

The program Π_{a_i} of each agent a_i is correct and complete. It is a positive program. Given a correct set of facts encoded by the predicate $table\_max\_a_c$ from the child agents of the agent a_i, it computes the correct aggregated utility table collected at agent a_i and then selects the optimal rows that will be sent to agent a_i's parent (predicate $table\_max\_a_i$). Likewise, given a set of correct solutions of the ancestor agents of a_i (predicate $solution$), it computes the correct solution for variables of agent a_i. Therefore, as long as the ASP solvers used by the agents is correct and complete, the correctness and completeness of ASP-DPOP follow quite trivially from that of DPOP since ASP-DPOP emulates the computation and communication operations of DPOP.

Each agent in ASP-DPOP, like DPOP, needs to compute, store, and send a utility for each value combination of its separator. Therefore, like DPOP, ASP-DPOP sends the same number of messages and also suffers from an exponential memory requirement—the memory requirement per agent is $O(maxDom^w)$, where $maxDom = \max_i |D_i|$ and w is the induced width of the pseudo-tree.

Related Work

The use of declarative programs, specifically logic programs, for reasoning in multi-agent domains is not new. Starting with some seminal papers (Kowlaski and Sadri 1999), various authors have explored the use of several different flavors of logic programming, such as normal logic programs and abductive logic programs, to address cooperation between agents (Kakas, Torroni, and Demetriou 2004; Sadri and Toni 2003; Gelfond and Watson 2007; De Vos et al. 2005). Some proposals have also explored the combination between constraint programming, logic programming, and formalization of multi-agent domains (Dovier, Formisano, and Pontelli 2013; Vlahavas 2002). Logic programming has been used in modeling multi-agent scenarios involving agents knowledge about other's knowledge (Baral et al. 2010), computing models in the logics of knowledge (Pontelli et al. 2010), multi-agent planning (Son, Pontelli, and Sakama 2009) and formalizing negotiation (Sakama, Son, and Pontelli 2011). ASP-DPOP is similar to the last two applications in that (*i*) it can be viewed as a collection of agent programs; (*ii*) it computes solutions using an ASP solver; and (*iii*) it uses message passing for agent communication. A key difference is that ASP-DPOP solves multi-agent problems formulated as constraint-based models, while the other applications solve problems formulated as decision-theoretic and game-theoretic models.

Researchers have also developed a framework that integrates declarative techniques with off-the-shelf constraint solvers to partition large constraint optimization problems into smaller subproblems and solve them in parallel (Liu et al. 2012). In contrast, DCOPs are problems that are naturally distributed and cannot be arbitrarily partitioned.

ASP-DPOP is able to exploit problem structure by propagating hard constraints and using them to prune the search space efficiently. This reduces the memory requirement of the algorithm and improves the scalability of the system. Existing DCOP algorithms that also propagates hard and soft constraints to prune the search space include H-DPOP that propagates exclusively hard constraints (Kumar, Petcu, and Faltings 2008), BrC-DPOP that propagates branch consistency (Fioretto et al. 2014), and variants of BnB-ADOPT (Yeoh, Felner, and Koenig 2010; Gutierrez and Meseguer 2012b; Gutierrez, Meseguer, and Yeoh 2011) that maintains soft-arc consistency (Bessiere, Gutierrez, and Meseguer 2012; Gutierrez and Meseguer 2012a; Gutierrez et al. 2013). A key difference is that these algorithms require algorithm developers to explicitly implement the ability to reason about the hard and soft constraints and propagate them efficiently. In contrast, ASP-DPOP capitalizes on general purpose ASP solvers to do so.

Experimental Results

We implement two versions of the ASP-DPOP algorithm—one that uses ground programs, which we call "ASP-DPOP (facts)," and one that uses non-ground programs, which we call "ASP-DPOP (rules)." In addition, we also implemented a variant of H-DPOP called PH-DPOP, which stands for Privacy-based H-DPOP, that restricts the amount of information that each agent can access to the amount common in most DCOP algorithms including ASP-DPOP. Specifically, agents in PH-DPOP can only access their own constraints and, unlike H-DPOP, cannot access their neighboring agents' constraints.

In our experiments, we compare both versions of ASP-DPOP against DPOP (Petcu and Faltings 2005), H-DPOP (Kumar, Petcu, and Faltings 2008), and PH-DPOP. We use a publicly-available implementation of DPOP (Léauté, Ottens, and Szymanek 2009) and an implementation of H-DPOP provided by the authors. We ensure that all algorithms use the same pseudo-tree for fair comparisons. We measure the runtime of the algorithms using the simulated runtime metric (Sultanik, Lass, and Regli 2007). All experiments are performed on a Quadcore 3.4GHz machine with 16GB of memory. If an algorithm fails to solve a problem, it is due to memory limitations. We conduct our experiments on random graphs (Erdös and Rényi 1959), where we systematically vary domain-independent parameters, and on comprehensive optimization problems in power networks (Gupta et al. 2013).

Random Graphs: In our experiments, we vary the number of variables $|\mathcal{X}|$, the domain size $|D_i|$, the constraint density

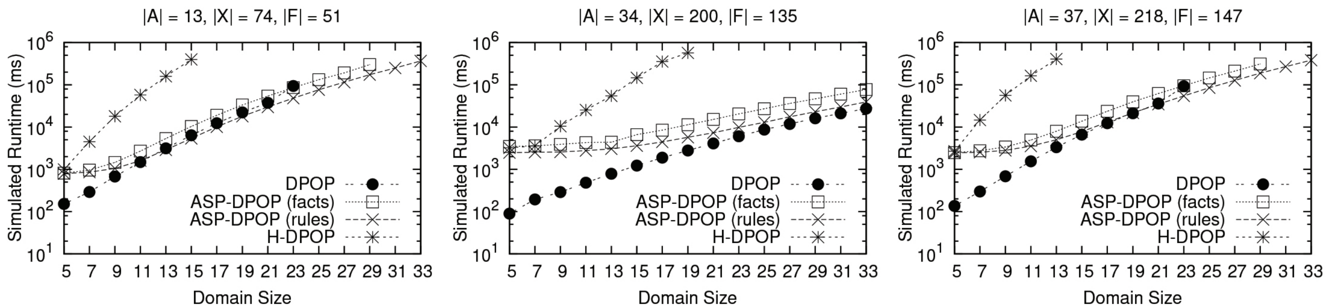
$\lvert\mathcal{X}\rvert$	DPOP		H-DPOP		PH-DPOP		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time
5	100%	36	100 %	28	100 %	31.47	100%	779
10	100%	204	100 %	73	100 %	381.02	100%	1,080
15	86%	39,701	100 %	148	98 %	67,161	100%	1,450
20	0%	-	100 %	188	0 %	-	100%	1,777
25	0%	-	100 %	295	0 %	-	100%	1,608

$\lvert\mathcal{D}_i\rvert$	DPOP		H-DPOP		PH-DPOP		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time
4	100%	782	100 %	87	100 %	1,512	100%	1,037
6	90%	28,363	100 %	142	98 %	42,275	100%	1,283
8	14%	37,357	100 %	194	52 %	262,590	100%	8,769
10	0%	-	100 %	320	8 %	354,340	100%	29,598
12	0%	-	100 %	486	0%	-	100%	60,190

p_1	DPOP		H-DPOP		PH-DPOP		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time
0.4	100%	1,856	100 %	119	100 %	2,117	100%	1,984
0.5	100%	13,519	100 %	120	100 %	19,321	100%	1,409
0.6	94%	42,010	100 %	144	100 %	54,214	100%	1,308
0.7	56%	66,311	100 %	165	88 %	131,535	100%	1,096
0.8	20%	137,025	100 %	164	62 %	247,335	100%	1,073

p_2	DPOP		H-DPOP		PH-DPOP		ASP-DPOP	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time
0.4	86%	48,632	100 %	265	84 %	107,986	86%	50,268
0.5	94%	38,043	100 %	161	96 %	71,181	92%	4,722
0.6	90%	31,513	100 %	144	98 %	68,307	100%	1,410
0.7	90%	39,352	100 %	128	100 %	49,377	100%	1,059
0.8	92%	40,526	100 %	112	100 %	62,651	100%	1,026

Table 2: Experimental Results on Random Graphs



(a) 13 Bus Topology (b) 34 Bus Topology (c) 37 Bus Topology

Figure 3: Runtime Results on Power Network Problems

p_1, and the constraint tightness p_2. For each experiment, we vary only one parameter and fix the rest to their "default" values: $|\mathcal{A}| = 5, |\mathcal{X}| = 15, |D_i| = 6, p_1 = 0.6, p_2 = 0.6$. Table 2 shows the percentage of instances solved (out of 50 instances) and the average simulated runtime (in ms) for the solved instances. We do not show the results for ASP-DPOP (rules), as the utilities in the utility table are randomly generated, leading to no differences w.r.t. ASP-DPOP (facts). We make the following observations:

- ASP-DPOP is able to solve more problems than DPOP and is faster than DPOP when the problem becomes more complex (i.e., increasing $|\mathcal{X}|$, $|D_i|$, p_1, or p_2). The reason is that ASP-DPOP is able to prune a significant portion of the search space thanks to hard constraints. ASP-DPOP does not need to explicitly represent the rows in the UTIL table that are infeasible, unlike DPOP. The size of the search space pruned increases as the complexity of the instance grows, resulting in a larger difference between the runtime of DPOP and ASP-DPOP.
- H-DPOP is able to solve more problems and solve them faster than every other algorithm. The reason is because agents in H-DPOP utilize more information about the neighbors' constraints to prune values. In contrast, agents in ASP-DPOP and PH-DPOP only utilize information about their own constraints to prune values and agents in DPOP do not prune any values.
- ASP-DPOP is able to solve more problems and solve them faster than PH-DPOP. The reason is that agents in PH-DPOP, like agents in H-DPOP, use constraint decision diagram (CDD) to represent their utility tables, and it is expensive to maintain and perform join and project operations on this data structure. In contrast, agents in ASP-DPOP is able to capitalize on highly efficient ASP solvers to maintain and perform operations on efficient data structures thanks to their highly optimized grounding techniques and use of portfolios of heuristics.

Power Network Problems: 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 transmission 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 ASP-DPOP (rules). For this reason, both "ASP-DPOP (facts)" and "ASP-DPOP (rules)' were used in this domain.

We use three network topologies defined using the IEEE

$\lvert\mathcal{D}_i\rvert$	13 Bus Topology				34 Bus Topology				37 Bus Topology			
	5	7	9	11	5	7	9	11	5	7	9	11
H-DPOP	6,742	30,604	97,284	248,270	1,437	4,685	11,617	24,303	6,742	30,604	97,284	248,270
DPOP	3,125	16,807	59,049	161,051	625	2,401	6,561	14,641	3,125	16,807	59,049	161,051
ASP-DPOP	10	14	18	22	10	14	18	22	10	14	18	22

(a) Largest UTIL Message Size

H-DPOP	19,936	79,322	236,186	579,790	20,810	57,554	130,050	256,330	38,689	133,847	363,413	836,167
DPOP	9,325	43,687	143,433	375,859	9,185	29,575	73,341	153,923	17,665	71,953	215,793	531,025
ASP-DPOP	120	168	216	264	330	462	594	726	360	504	648	792

(b) Total UTIL Message Size

Table 3: Message Size Results on Power Network Problems

standards (IEEE Distribution Test Feeders 2014) and vary the domain size of the generation, load, and transmission variables of each agent from 5 to 31. Figure 3 summarizes the runtime results. As the utilities are generated following predefined rules (Gupta et al. 2013), we also show the results for ASP-DPOP (rules). Furthermore, we omit results for PH-DPOP because they have identical runtime—the amount of information used to prune the search space is identical for both algorithms in this domain. We also measure the size of UTIL messages, where we use the number of values in the message as units. Table 3 tabulates the results. We did not measure the size of VALUE messages since they are significantly smaller than UTIL messages.

The results in Figure 3 are consistent with those shown earlier—ASP-DPOP is slower than DPOP when the domain size is small, but it is able to solve more problems than DPOP. We observe that, in Figure 3(b), DPOP is consistently faster than ASP-DPOP and is able to solve the same number of problems as ASP-DPOP. It is because the highest constraint arity in 34 bus topology is 5 while it is 6 in 13 and 37 bus topologies. Unlike in random graphs, H-DPOP is slower than the other algorithms in these problems. The reason is that the constraint arity in these problems are larger and the expensive operations on CDDs grows exponentially with the arity. We also observe that ASP-DPOP (rules) is faster than ASP-DPOP (facts). The reason is that the former is able to exploit the interdependencies between constraints to prune the search space. Additionally, ASP-DPOP (rules) can solve more problems than ASP-DPOP (facts). The reason is that the former requires less memory since it prunes a larger search space and, thus, ground fewer facts. Finally, both versions of ASP-DPOP require smaller messages than both H-DPOP and DPOP. The reason for the former is that the CDD data structure of H-DPOP is significantly more complex than that of ASP-DPOP, and the reason for the latter is because ASP-DPOP pruned portions of the search space while DPOP did not.

Conclusions

In this paper, we explore the new direction of DCOP algorithms that use logic programming techniques. Our proposed logic-programming-based algorithm, ASP-DPOP, is able to solve more problems and solve them faster than DPOP, its imperative programming counterpart. Aside from the ease of modeling, each agent in ASP-DPOP also capitalizes on highly efficient ASP solvers to automatically exploit problem structure (e.g., prune the search space using hard constraints). Experimental results show that ASP-DPOP is faster and can scale to larger problems than a version of H-DPOP that limits its knowledge to the same amount as ASP-DPOP. These results highlight the strength of the declarative programming paradigm, where explicit model-specific pruning rules are not necessary. In conclusion, we believe that this work contributes to the DCOP community, where we show that the declarative programming paradigm is a promising new direction of research for DCOP researchers, as well as the ASP community, where we demonstrate the applicability of ASP to solve a wide array of multi-agent problems that can be modeled as DCOPs.

Acknowledgment

This research is partially supported by NSF grants HRD-1345232 and DGE-0947465. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government. We would like to thank Akshat Kumar for sharing with us his implementation of H-DPOP.

References

Baral, C.; Gelfond, G.; Pontelli, E.; and Son, T. C. 2010. Modeling multi-agent scenarios involving agents knowledge about other's knowledge using ASP. In *Proc. of AAMAS*, 259–266.

Bessiere, C.; Gutierrez, P.; and Meseguer, P. 2012. Including soft global constraints in DCOPs. In *Proc. of CP*, 175–190.

Citrigno, S.; Eiter, T.; Faber, W.; Gottlob, G.; Koch, C.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. The dlv system: Model generator and application frontends. In *Proc. of the Workshop on Logic Programming*, 128–137.

De Vos, M.; Crick, T.; Padget, J. A.; Brain, M.; Cliffe, O.; and Needham, J. 2005. LAIMA: A multi-agent platform using ordered choice logic programming. In *Proc. of DALT*.

Dovier, A.; Formisano, A.; and Pontelli, E. 2013. Autonomous agents coordination: Action languages meet CLP() and Linda. *Theory and Practice of Logic Programming* 13(2):149–173.

Erdös, P., and Rényi, A. 1959. On random graphs i. *Publicationes Mathematicae Debrecen* 6:290.

Farinelli, A.; Rogers, A.; Petcu, A.; and Jennings, N. 2008. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proc. of AAMAS*, 639–646.

Fioretto, F.; Le, T.; Yeoh, W.; Pontelli, E.; and Son, T. C. 2014. Improving DPOP with branch consistency for solving distributed constraint optimization problems. In *Proc. of CP*.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. clasp: A conflict-driven answer set solver. In *Proc. of LPNMR*, 260–265.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proc. of ICLP*, 579–597.

Gelfond, G., and Watson, R. 2007. Modeling cooperative multi-agent systems. In *Proc. of ASP Workshop*.

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 *Proc. of the Distributed Constraint Reasoning Workshop*, 45–59.

Gutierrez, P., and Meseguer, P. 2012a. Improving BnB-ADOPT$^+$-AC. In *Proc. of AAMAS*, 273–280.

Gutierrez, P., and Meseguer, P. 2012b. Removing redundant messages in n-ary BnB-ADOPT. *Journal of Artificial Intelligence Research* 45:287–304.

Gutierrez, P.; Lee, J.; Lei, K. M.; Mak, T.; and Meseguer, P. 2013. Maintaining soft arc consistencies in BnB-ADOPT$^+$ during search. In *Proc. of CP*, 365–380.

Gutierrez, P.; Meseguer, P.; and Yeoh, W. 2011. Generalizing ADOPT and BnB-ADOPT. In *Proc. of IJCAI*, 554–559.

Hamadi, Y.; Bessière, C.; and Quinqueton, J. 1998. Distributed intelligent backtracking. In *Proc. of ECAI*, 219–223.

IEEE Distribution Test Feeders. 2014. http://ewh.ieee.org/soc/pes/dsacom/testfeeders/.

Jain, P.; Gupta, S.; Ranade, S.; and Pontelli, E. 2012. Optimum operation of a customer-driven microgrid: A comprehensive approach. In *Proc. of PEDES*.

Kakas, A.; Torroni, P.; and Demetriou, N. 2004. Agent Planning, negotiation and control of operation. In *Proc. of ECAI*.

Kowlaski, R., and Sadri, F. 1999. Logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence* 25(3-4):391–419.

Kumar, A.; Faltings, B.; and Petcu, A. 2009. Distributed constraint optimization with structured resource constraints. In *Proc. of AAMAS*, 923–930.

Kumar, A.; Petcu, A.; and Faltings, B. 2008. H-DPOP: Using hard constraints for search space pruning in DCOP. In *Proc. of AAAI*, 325–330.

Lass, R.; Kopena, J.; Sultanik, E.; Nguyen, D.; Dugan, C.; Modi, P.; and Regli, W. 2008. Coordination of first responders under communication and resource constraints (Short Paper). In *Proc. of AAMAS*, 1409–1413.

Léauté, T., and Faltings, B. 2011. Coordinating logistics operations with privacy guarantees. In *Proc. of IJCAI*, 2482–2487.

Léauté, T.; Ottens, B.; and Szymanek, R. 2009. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proc. of the Distributed Constraint Reasoning Workshop*, 160–164.

Liu, C.; Ren, L.; Loo, B. T.; Mao, Y.; and Basu, P. 2012. Cologne: A declarative distributed constraint optimization platform. *Proceedings of the VLDB Endowment* 5(8):752–763.

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 *Proc. of AAMAS*, 310–317.

Mailler, R., and Lesser, V. 2004. Solving distributed constraint optimization problems using cooperative mediation. In *Proc. of AAMAS*, 438–445.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, 375–398.

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 *Proc. of AAMAS*, 167–174.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3–4):241–273.

Ottens, B.; Dimitrakakis, C.; and Faltings, B. 2012. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proc. of AAAI*, 528–534.

Petcu, A., and Faltings, B. 2005. A scalable method for multiagent constraint optimization. In *Proc. of IJCAI*, 1413–1420.

Pontelli, E.; Son, T. C.; Baral, C.; and Gelfond, G. 2010. Logic programming for finding models in the logics of knowledge and its applications: A case study. *Theory and Practice of Logic Programming* 10(4-6):675–690.

Sadri, F., and Toni, F. 2003. Abductive logic programming for communication and negotiation amongst agents. ALP Newsletter.

Sakama, C.; Son, T. C.; and Pontelli, E. 2011. A logical formulation for negotiation among dishonest agents. In *Proc. of IJCAI*, 1069–1074.

Son, T. C.; Pontelli, E.; and Sakama, C. 2009. Logic programming for multiagent planning with negotiation. In *Proc. of ICLP*, 99–114.

Sultanik, E.; Lass, R.; and Regli, W. 2007. DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proc. of the Distributed Constraint Reasoning Workshop*.

Ueda, S.; Iwasaki, A.; and Yokoo, M. 2010. Coalition structure generation based on distributed constraint optimization. In *Proc. of AAAI*, 197–203.

Vlahavas, I. 2002. MACLP: Multi Agent Constraint Logic Programming. *Information Sciences* 144(1-4):127–142.

Yeoh, W., and Yokoo, M. 2012. Distributed problem solving. *AI Magazine* 33(3):53–65.

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.

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.; Glinton, R.; and Sycara, K. 2009. Distributed constraint optimization for large teams of mobile sensing agents. In *Proc. of IAT*, 347–354.

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