ODPOP: An Algorithm For Open/Distributed Constraint Optimization

Adrian Petcu and **Boi Faltings** *
{adrian.petcu, boi.faltings}@epfl.ch
Artificial Intelligence Laboratory (http://liawww.epfl.ch/)
Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

Abstract

We propose ODPOP, a new distributed algorithm for *open multiagent combinatorial optimization* that feature unbounded domains (Faltings & Macho-Gonzalez 2005). The ODPOP algorithm explores the same search space as the dynamic programming algorithm DPOP (Petcu & Faltings 2005b) or ADOPT (Modi *et al.* 2005), but does so in an incremental, best-first fashion suitable for open problems.

ODPOP has several advantages over DPOP. First, it uses messages whose size only grows linearly with the treewidth of the problem. Second, by letting agents explore values in a best-first order, it avoids incurring always the worst case complexity as DPOP, and on average it saves a significant amount of computation and information exchange.

To show the merits of our approach, we report on experiments with practically sized distributed meeting scheduling problems in a multiagent system.

Introduction

Constraint satisfaction and optimization is a powerful paradigm for solving numerous tasks in distributed AI, like planning, scheduling, resource allocation, etc. Many real problems are naturally distributed among a set of agents, each one holding its own subproblem. The agents have to communicate with each other to find an optimal solution to the overall problem (unknown to any one of them). In such settings, centralized optimization algorithms are often unsuitable because it may be unpractical or even impossible to gather the whole problem into a single place. Distributed Constraint Satisfaction (DisCSP) has been formalized by Dechter (Collin, Dechter, & Katz 1991), Meisels (Solotorevsky, Gudes, & Meisels 1996) and Yokoo (Yokoo *et al.* 1998) to address such problems.

Complete algorithms for distributed constraint optimization fall in two main categories: search (see (Collin, Dechter, & Katz 1991; Yokoo *et al.* 1998; Silaghi, Sam-Haroud, & Faltings 2000; Modi *et al.* 2005; Hamadi, Bessière, & Quinqueton 1998)), and dynamic programming (see (Petcu & Faltings 2005b; Kask, Dechter, & Larrosa 2005)).

*This work has been funded by the Swiss National Science Foundation under contract No. 200020-103421/1.

Search algorithms require linear memory and message size, and the worst case complexity can sometimes be avoided if effective pruning is possible. However, they produce an exponential number of small messages, which typically entails large networking overheads.

Dynamic programming algorithms have the important advantage that they produce fewer messages, therefore less overhead. DPOP (Petcu & Faltings 2005b) for example requires a linear number of messages. The disadvantage is that the maximal message size and memory requirements grows exponentially in the induced width of the constraint graph. Furthermore, the worst case complexity is always incurred.

This paper presents ODPOP, which combines some advantages of both worlds: it does not always incur the worst case complexity, it always uses linear size messages (as with search), and typically generates few messages (as DPOP).

Although its worst case complexity is the same as for DPOP, ODPOP typically exhibits significant savings in computation and information exchange. This is because the agents in ODPOP use a best-first order for value exploration, and an optimality criterion that allows them to prove optimality even without exploring all the values of their parents.

This makes ODPOP applicable also to open constraint optimization problems, where variables may have unbounded domains (Faltings & Macho-Gonzalez 2005).

We introduce the DCOP problem, definitions and notations, and the basic DPOP algorithm. We then describe the ODPOP algorithm, show examples, and evaluate its complexity, both theoretically and experimentally. We present experimental results on meeting scheduling problems, and then conclude.

Definitions and Notation

Definition 1 *A discrete* distributed constraint optimization problem *(DCOP) is a tuple* $< \mathcal{X}, \mathcal{D}, \mathcal{R} >$ *such that:*

- $\mathcal{X} = \{x_1, ..., x_n\}$ *is a set of variables*
- $\mathcal{D} = \{d_1, ..., d_n\}$ *is a set of finite domains of the variables*
- $\mathcal{R} = \{r_1, ..., r_m\}$ *is a set of relations, where a relation* r_i *is any function with the scope* $(x_{i_1}, \cdots, x_{i_k})$, $r_i : d_{i_1} \times .. \times d_{i_k} \rightarrow \mathbb{R}$*, which denotes how much utility is assigned to each possible combination of values of the involved variables. Negative amounts mean costs.* [1]

[1] Hard constraints (that explicitly forbid/enforce certain value

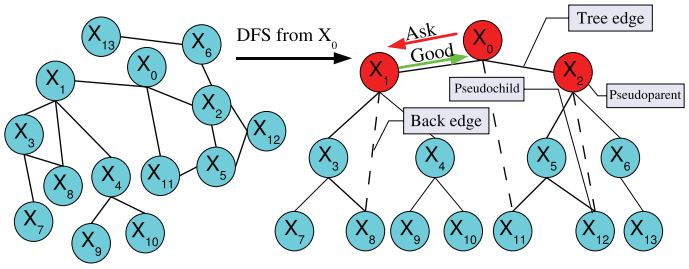

Figure 1: *A problem graph and a rooted DFS tree. ASK messages go top-down, and GOOD messages (valued goods) go bottom-up. All messages are of linear size.*

In a DCOP, each variable and constraint is owned by an agent. A simplifying assumption (Yokoo *et al.* 1998) is that each agent controls a virtual agent for each one of the variables x_i that it owns. To simplify the notation, we use x_i to denote either the variable itself, or its (virtual) agent.

This is a multiagent instance of the *valued CSP* framework as defined by Schiex et al (Schiex, Fargier, & Verfaillie 1995). The goal is to find a complete instantiation $\mathcal{X}^*$ for the variables x_i that *maximizes* the aggregate utility, i.e. the sum of utilities of individual relations. We assume here only unary and binary constraints/relations. DPOP and ODPOP extend however easily to non-binary constraints (see (Petcu, Faltings, & Parkes 2006)).

Depth-First Search Trees (DFS)

ODPOP works on a DFS traversal of the problem graph.

Definition 2 *A DFS arrangement of a graph G is a rooted tree with the same nodes and edges as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g. x_0 and x_{11} in Figure 1).*

DFS trees have already been investigated as a means to boost search (Freuder 1985; Dechter 2003). Due to the relative independence of nodes lying in different branches of the DFS tree, it is possible to perform search in parallel on these independent branches.

Figure 1 shows an example of a DFS tree that we shall refer to in the rest of this paper. We distinguish between *tree edges*, shown as solid lines (e.g. $8-3$), and *back edges*, shown as dashed lines (e.g. $8-1, 12-2$).

Definition 3 (DFS concepts) *Given a node x_i, we define:*

- *the* **parent** P_i **/ children** C_i*: these are the obvious definitions (e.g. $P_4 = x_1$, $C_1 = \{x_3, x_4\}$).*
- *The* **pseudo-parents** PP_i *are x_i's ancestors that are connected to x_i directly through back-edges ($PP_8 = \{x_1\}$).*
- *The* **pseudo-children** PC_i *are x_i's descendents directly connected to x_i through back-edges (e.g. $PC_0 = \{x_{11}\}$).*
- *Sep_i is the* **separator** *of x_i: ancestors of x_i which are directly connected with x_i or with descendants of x_i (e.g. $Sep_4 = \{x_1\}$, $Sep_5 = \{x_0, x_2\}$ and $Sep_8 = \{x_1, x_3\}$).*

combinations) can be simulated with soft constraints by assigning $-\infty$ to disallowed tuples, and 0 to allowed tuples. Maximizing utility thus avoids assigning such value combinations to variables.

Sep_i is the set of ancestors of x_i whose removal completely disconnects the subtree rooted at x_i from the rest of the problem. In case the problem is a tree, then $Sep_i = \{P_i\}, \forall x_i \in \mathcal{X}$. In the general case, Sep_i contains P_i, all PP_i and all the pseudoparents of all descendants of x_i, which are ancestors of x_i.

DPOP: dynamic programming optimization

DPOP is a distributed version of the bucket elimination scheme from (Dechter 2003), which works on a DFS. *DPOP* has 3 phases:

Phase 1 - a **DFS traversal** of the graph is done using a distributed DFS algorithm. To save space, we refer the reader to an algorithm like (Petcu, Faltings, & Parkes 2006). The outcome of this protocol is that all nodes consistently label each other as parent/child or pseudoparent/pseudochild, and edges are identified as tree/back edges.

Phase 2 - **UTIL propagation** is a bottom-up process, which starts from the leaves and propagates upwards only through tree edges. The agents send *UTIL* messages to their parents. The subtree of a node X_i can influence the rest of the problem only through X_i's separator, Sep_i. Therefore, a message contains the optimal utility obtained in the subtree for each instantiation of Sep_i. Thus, messages are exponential in the separator size (bounded by the induced width).

Phase 3 - **VALUE propagation** top-down, initiated by the root, when phase 2 has finished. Each node determines its optimal value based on the computation from phase 2 and the *VALUE* message it has received from its parent. Then, it sends this value to its children through *VALUE* messages.

ODPOP: an optimization algorithm for DCOP

ODPOP is described in Algorithm 1. It also has 3 phases:

Phase 1 - a **DFS traversal** as in DPOP

Phase 2 - (**ASK/GOOD**) this is where ODPOP is different from DPOP. This is an iterative, bottom-up utility propagation process, where each node repeatedly asks (via *ASK* messages) its children for valuations (*goods*) until it can compute suggested optimal values for its ancestors included in its separator. It then sends these goods to its parent. This phase finishes when the root received enough valuations to determine its optimal value.

Phase 3 - **VALUE propagation** as in DPOP

ODPOP Phase 2: *ASK/GOOD* Phase

In backtracking algorithms, the control strategy is top-down: starting from the root, the nodes perform assignments and inform their children about these assignments. In return, the children determine their best assignments given these decisions, and inform their parents of the utility or bounds on this utility.

This top-down exploration of the search space has the disadvantage that the parents make decisions about their values blindly, and need to determine the utility for every one of their values before deciding on the optimal one. This can be a very costly process, especially when domains are large.

Additionally, if memory is bounded, many utilities have to be derived over and over again (Modi *et al.* 2005). This,

Algorithm 1 *ODPOP - Open/Distributed Optimization*

ODPOP($\mathcal{X}, \mathcal{D}, \mathcal{R}$): each agent x_i does:

DFS arrangement: run token passing mechanism as in (Petcu, Faltings, & Parkes 2006)

At completion, x_i knows $P_i, PP_i, C_i, PC_i, Sep_i$

Main process

$sent\_goods \leftarrow \emptyset$

if x_i *is root* **then**

ASK/GOOD until valuation sufficiency

else

while *!received VALUE message* **do**

Process incoming ASK and GOOD messages

Process ASK

while *!sufficiency conditional on* $sent\_goods$ **do**

select C_i^{ask} among C_i

send ASK message to all C_i^{ask}

wait for GOOD messages

find $best\_good \in Sep_i$ s.t. $best\_good \notin sent\_goods$

add $best\_good$ to $sent\_goods$, and send it to P_i

Process GOOD$(\mathbf{gd}, \mathbf{x_k})$

add gd to $goodstore(x_k)$

check for conditional sufficiency

coupled with the asynchrony of these algorithms makes for a large amount of effort to be duplicated unnecessarily (Zivan & Meisels 2004).

Propagating GOODs In contrast, we propose a bottom-up strategy in ODPOP, similar to the one of DPOP. In this setting, higher nodes do not assign themselves values, but instead ask their children what values would be best. Children answer by proposing values for the parents' variables. Each such proposal is called a *good*, and has an associated utility that can be achieved by the subtree rooted at the child, in the context of the proposal.

Definition 4 (Good) *Given a node x_i, its parent P_i and its separator Sep_i, a* **good message** $GOOD_i^{P_i}$ *sent from x_i to P_i is a tuple $\langle assignments, utility\rangle$ as follows: $GOOD_i^{P_i} = \langle \{x_j = v_j^k | x_j \in Sep_i, v_j^k \in D_j\}, v \in \mathbb{R}\rangle$.*

In words, a good $GOOD_i^{P_i}$ sent by a node x_i to its parent P_i has exactly one assignment for each variable in Sep_i, plus the associated utility generated by this assignment for the subtree rooted at x_i. In the example of Figure 1, a good sent from x_5 to x_2 might have this form: $GOOD_5^2 = \langle x_2 = a, x_0 = c, 15\rangle$, which means that if $x_2 = a$ and $x_0 = c$, then the subtree rooted at x_5 gets 15 units of utility.

Definition 5 (Compatibility: $\equiv$) *Two good messages $GOOD_1$ and $GOOD_2$ are* **compatible** *(we write this $GOOD_1 \equiv GOOD_2$) if they do not differ in any assignment of the shared variables. Otherwise, $GOOD_1 \not\equiv GOOD_2$.*

Example: $\langle x_2 = a, x_0 = c, 15\rangle \equiv \langle x_2 = a, 7\rangle$, but $\langle x_2 = a, x_0 = c, 15\rangle \not\equiv \langle x_2 = b, 7\rangle$.

Definition 6 (Join: $\oplus$) *The* **join** $\oplus$ *of two* compatible *good messages $GOOD_j^i = \langle assig_j, val_j\rangle$ and $GOOD_k^i = \langle assig_k, val_k\rangle$ is a new good $GOOD_{j,k}^i = \langle assig_j \cup assig_k, val_j + val_k\rangle$*

Example in Figure 1: let $GOOD_{11}^5 = \langle x_5 = a, x_0 = c, 15\rangle$ and $GOOD_{12}^5 = \langle x_5 = a, x_2 = b, 7\rangle$. Then $GOOD_{11}^5 \oplus GOOD_{12}^5 = \langle x_2 = b, x_0 = c, x_5 = a, 22\rangle$.

Value ordering and bound computation Any child x_j of a node x_i delivers to its parent x_i a sequence of $GOOD_j^i$ messages that explore different combinations of values for the variables in Sep_j, together with the corresponding utilities. We introduce the following important assumption:

Best-first Assumption: leaf nodes (without children) report their $GOOD$s in order of non-increasing utility.

This assumption is easy to satisfy in most problems: it corresponds to ordering entries in a relation according to their utilities. Similarly, agents usually find it easy to report what their most preferred outcomes are.

We now show a method for propagating $GOOD$s so that all nodes always report $GOOD$s in order of non-increasing utility provided that their children follow this order. Together with the assumption above, this will give an algorithm where the first $GOOD$ generated at the root node is the optimal solution. Furthermore, the algorithm will be able to generate this solution without having to consider all value combinations.

Consider thus a node x_i that receives from each of its children x_j a stream of $GOOD$s in an asynchronous fashion, but in non-increasing order of utility.

Notation: let $LAST_j^i$ be the last good sent by x_j to x_i. Let $\langle Sep_i\rangle$ be the set of all possible instantiations of variables in Sep_i. A tuple $s \in \langle Sep_i\rangle$ is such an instantiation. Let $GOOD_j^i(t)$ be a good sent by x_j to x_i that is compatible with the assignments in the tuple t.

Based on the goods that x_j has already sent to x_i, one can define lower (LB) and upper (UB) bounds for each instantiation $s \in \langle Sep_i\rangle$:

$$LB_j^i(s) = \begin{cases} val(GOOD_j^i(t)) & \text{if } x_j \text{ sent } GOOD_j^i(t) \text{ s.t. } t \equiv s \\ -\infty & \text{otherwise} \end{cases}$$

$$UB_j^i(s) = \begin{cases} val(GOOD_j^i(t)) & \text{if } x_j \text{ sent } GOOD_j^i(t) \text{ s.t. } t \equiv s \\ val(LAST_j^i) & \text{if } x_j \text{ has sent any } GOOD_j^i \\ +\infty & \text{if } x_j \text{ has not sent any } GOOD_j^i \end{cases}$$

The influence of all children of x_i is combined in upper and lower bounds for each $s \in \langle Sep_i\rangle$ as follows:

- $UB^i(s) = \sum_{x_j \in C_i} UB_j^i(s)$; if any of $x_j \in C_i$ has not yet sent any good, then $UB_j^i(s) = +\infty$, and $UB^i(s) = +\infty$. $UB^i(s)$ is the maximal utility that the instantiation s could possibly have for the subproblem rooted at x_i, no matter what other goods will be subsequently received by x_i. Note that it is possible to infer an upper bound on the utility of any instantiation $s \in \langle Sep_i\rangle$ as soon as even a single $GOOD$ message has been received from each child. This is the result of the assumption that $GOOD$s are reported in order of non-increasing utility.
- $LB^i(s) = \sum_{x_j \in C_i} LB_j^i(s)$; if any of $x_j \in C_i$ has not yet sent any good compatible with s, then $LB_j^i(s) = -\infty$,

and $LB^i(s) = -\infty$. $LB^i(s)$ is the minimal utility that the tuple $s \in \langle Sep_i \rangle$ could possibly have for the subproblem rooted at x_i, no matter what other goods will be subsequently received by x_i.

Examples based on Table 2:

- $GOOD^4_{10}(x_4 = c) = \langle [x_4 = c], 4 \rangle$.
- $LAST^4_{10} = \langle [x_4 = a], 3 \rangle$.
- $LB^4_{10}(x_4 = c) = 4$ and $LB^4_9(x_4 = c) = -\infty$, because x_4 has received a $GOOD^4_{10}(x_4 = c)$ from x_{10}, but not a $GOOD^4_9(x_4 = c)$ from x_9.
- Similarly, $UB^4_{10}(x_4 = c) = 4$ and $UB^4_9(x_4 = c) = val(LAST^4_9) = val(GOOD^4_9(x_4 = f)) = 1$, because x_4 has received a $GOOD(x_4 = c)$ from x_{10}, but not from x_9, so the latter is replaced by the latest received good.

Valuation-Sufficiency

In DPOP, agents receive all $GOOD$s grouped in single messages. Here, $GOOD$s can be sent individually and asynchronously as long as the order assumption is satisfied. Therefore, x_i can determine when it has received **enough** goods from its children in order to be able to determine the next best combination of values of variables in Sep_i (Faltings & Macho-Gonzalez 2005). In other words, x_i can determine when any additional goods received from its children x_j will not matter w.r.t. the choice of optimal tuple for Sep_i. x_i can then send its parent P_i a valued good $t^* \in Sep_i$ suggesting this next best value combination.

Definition 7 *Given a subset S of tuples from $\langle Sep_i \rangle$, a tuple $t^* \in \{\langle Sep_i \rangle \setminus S\}$ is* dominant conditional on the subset S, *when $\forall t \in \{\langle Sep_i \rangle \setminus S | t \neq t^*\}, LB(t^*) > UB(t)$.*

In words, t^* is the next best choice for Sep_i, after the tuples in S. This can be determined once there have been received enough goods from children to allow the finding that one tuple's lower bound is greater than all other's upper bound. Then the respective tuple is conditional-dominant.

Definition 8 *A variable is* valuation-sufficient conditional on a subset $S \subset \langle Sep_i \rangle$ of instantiations of the separator *when it has a tuple t^* which is dominant conditional on S.*

Properties of the Algorithm The algorithm used for propagating $GOOD$s in ODPOP is given by process ASK in Algorithm 1. Whenever a new $GOOD$ is asked by the parent, x_i repeatedly asks its children for $GOOD$s. In response, it receives $GOOD$ messages that are used to update the bounds. These bounds are initially set to $LB^i(\forall t) = -\infty$ and $UB^i(\forall t) = +\infty$. As soon as at least one message has been received from all children for a tuple t, its upper bound is updated with the sum of the utilities received. As more and more messages are received, the bounds become tighter and tighter, until the lower bound of a tuple t^* becomes higher than the upper bound of any other tuple.

At that point, we call t^* *dominant*. x_i assembles a good message $GOOD^{P_i}_i = \langle t^*, val = LB^i(t^*) = UB^i(t^*) \rangle$, and sends it to its parent P_i. The tuple t^* is added to the $sent\_goods$ list.

$x_1/x_4 =$	a	b	c	d	e	f
$x_1 = a$	1	2	6	2	1	2
$x_1 = b$	5	1	2	1	2	1
$x_1 = c$	2	1	1	1	2	1

Table 1: Relation $R(x_4, x_1)$.

x_9	x_{10}	x_1
$\langle \mathbf{x_4 = a, 6} \rangle$	$\langle x_4 = b, 5 \rangle$	$\langle x_4 = c, x_1 = a, 6 \rangle$
$\langle x_4 = d, 5 \rangle$	$\langle x_4 = c, 4 \rangle$	$\langle \mathbf{x_4 = a, x_1 = b, 5} \rangle$
$\langle x_4 = f, 1 \rangle$	$\langle \mathbf{x_4 = a, 3} \rangle$	$\langle x_4 = b, x_1 = a, 2 \rangle$
$\vdots$	$\vdots$	$\vdots$

Table 2: Goods received by x_4. The relation r^1_4 is present in the last column, sorted best-first.

Subsequent ASK messages from P_i will be answered using the same principle: gather goods, recompute upper/lower bounds, and determine when another tuple is dominant. However, the dominance decision is made while ignoring the tuples from $sent\_goods$, so the "next-best" tuple will be chosen. This is how it is ensured that each node in the problem will receive utilities for tuples *in decreasing order of utility* i.e. in a best-first order, and thus we have the following Theorem:

Theorem 1 (Best-first order) *Provided that the leaf nodes order their relations in non-increasing order of utility, each node in the problem sends $GOOD$s in the non-increasing order of utility i.e. in a best-first order.*

PROOF. By assumption, the leaf nodes send $GOOD$s in best-first order. Assume that all children of x_i satisfy the Theorem. Then the algorithm correctly infers the upper bounds on the various tuples, and correctly decides conditional valuation-sufficiency. If it sends a $GOOD$, it is conditionally dominant given all $GOOD$s that were sent earlier, and so it cannot have a lower utility than any $GOOD$ that might be sent later. □

Conditional valuation-sufficiency: an example Let us consider a possible execution of ODPOP on the example problem from Figure 1. Let us consider the node x_4, and let the relation r^1_4 be as described in Table 1.

As a result to its parent x_1 asking x_4 for goods, let us assume that x_4 has repeatedly requested goods from its children x_9 and x_{10}. x_9 and x_{10} have replied each with goods; the current status is as described in Table 2.

In addition to the goods obtained from its children, x_4 has access to the relation r^1_4 with its parent, x_1. This relation will also be explored in a best-first fashion, exactly as the tuples received from x_4's children (see Table 2, last column).

Let us assume that this is the first time x_1 has asked x_4 for goods, so the $sent\_goods$ list is empty.

We compute the lower and upper bounds as described in the previous section. We obtain that $LB^i(\langle x_4 = a, x_1 = b \rangle) = 14$. We also obtain that $\forall t \neq \langle x_4 = a, x_1 = b \rangle$,

$UB^i(t) < LB^i(\langle x_4 = a, x_1 = b\rangle) = 14$. Therefore, $\langle x_4 = a, x_1 = b\rangle$ satisfies the condition from Definition 8 and is thus dominant conditional on the current $sent\_goods$ set (which is empty). Thus, x_4 records $\langle x_4 = a, x_1 = b, 14\rangle$ in $sent\_goods$ and sends $GOOD(x_1 = b, 14)$ to x_1.

Should x_1 subsequently ask for another good, x_4 would repeat the process, this time ignoring the previously sent tuple $GOOD(x_1 = b, 14)$.

Comparison with the *UTIL* phase of *DPOP* In DPOP, the separator Sep_i of a node x_i gives the set of dimensions of the *UTIL* message from x_i to its parent: $Sep_i = dims(UTIL_i^{P_i})$ Therefore, the size of a *UTIL* message in DPOP is $d^{|Sep_i|}$, where d is the domain size. This results in memory problems in case the induced width of the constraint graph is high.

In ODPOP, the *ASK/GOOD* phase is the analogue of the *UTIL* phase from *DPOP*. A $GOOD_i^{P_i}$ message corresponds exactly to a single utility from a $UTIL_i^{P_i}$ message from DPOP, and has the same semantics: it informs P_i how much utility the whole subtree rooted at x_i obtains when the variables from Sep_i take that particular assignment.

The difference is that the utilities are sent on demand, in an incremental fashion. A parent P_i of a node x_i sends to x_i an *ASK* message that instructs x_i to find the next best combination of values for the variables in Sep_i, and compute its associated utility. x_i then performs a series of the same kind of queries to its children, until it gathers enough goods to be able to determine this next best combination $t^* \in \langle Sep_i\rangle$ to send to P_i. At this point, x_i assembles a message $GOOD_i^{P_i}(t^*, val)$ and sends it to P_i.

ODPOP Phase 3: VALUE assignment phase

The *VALUE* phase is similar to the one from *DPOP*. Eventually, the root of the DFS tree becomes *valuation-sufficient*, and can therefore determine its optimal value. It initiates the top-down *VALUE* propagation phase by sending a *VALUE* message to its children, informing them about its chosen value. Subsequently, each node x_i receives the $VALUE_{P_i}^i$ message from its parent, and determines its optimal value as follows:

1. x_i searches through its $sent\_list$ for the first good $GOOD^{i*}$ (highest utility) compatible with the assignments received in the *VALUE* message.
2. x_i assigns itself its value from $GOOD^{i*}$: $x_i \leftarrow v_i^*$
3. $\forall x_j \in C_i$, x_i builds and sends a *VALUE* message that contains $x_i = v_i^*$ and the assignments shared between $VALUE_{P_i}^i$ and Sep_j. Thus, x_j can in turn choose its own optimal value, and so on recursively to the leaves.

ODPOP: soundness, termination, complexity

Theorem 2 (Soundness) *ODPOP is sound.*

PROOF. ODPOP combines goods coming from independent parts of the problem (subtrees in DFS are independent). Theorem 1 shows that the goods arrive in the best-first order, so when we have valuation-sufficiency, we are certain to choose the optimal tuple, provided the tuple from Sep_i is optimal.

The top-down *VALUE* propagation ensures (through induction) that the tuples selected to be parts of the overall optimal assignment, are indeed optimal, thus making also all assignments for all Sep_i optimal. □

Theorem 3 (Termination) *ODPOP terminates in at most $(h-1) \times d^w$ synchronous ASK/GOOD steps, where h is the depth of the DFS tree, d bounds the domain size, and w is the width of the chosen DFS. Synchronous here means that all siblings send their messages at the same time.*

PROOF. The longest branch in the DFS tree is of length $h-1$ (and h is at most n, when the DFS is a chain). Along a branch, there are at most d^{Sep_i} *ASK/GOOD* message pairs exchanged between any node x_i and its parent. Since $Sep_i \leq w$, it follows that at most $(h-1) \times d^w$ synchronous *ASK/GOOD* message pairs will be exchanged. □

Theorem 4 (Complexity) *The number of messages and memory required by ODPOP is $O(d^w)$.*

PROOF. By construction, all messages in ODPOP are linear in size. Regarding the number of messages:

1. the DFS construction phase produces a linear number of messages: $2 \times m$ messages (m is the number of edges);
2. the *ASK/GOOD* phase is the analogue of the *UTIL* phase in DPOP. The worst case behavior of ODPOP is to send sequentially the contents of the *UTIL* messages from DPOP, thus generating at most d^w *ASK/GOOD* message pairs between any parent/child node (d is the maximal domain size, and w is the induced width of the problem graph). Overall, the number of messages is $O((n-1) \times d^w)$. Since all these messages have to be stored by their recipients, the memory consumption is also at most d^w.
3. the *VALUE* phase generates $n-1$ messages, (n is the number of nodes) - one through each tree-edge.

□

Notice that the d^w complexity is incurred only in the worst case. Consider an example: a node X_i receives first from all its children the same tuple as their most preferred one. Then this is simply chosen as the best and sent forward, and X_i needs only linear memory and computation!

Experimental Evaluation

We experimented with distributed meeting scheduling in an organization with a hierarchical structure (a tree with departments as nodes, and a set of agents working in each department). The CSP model is the PEAV model from (Maheswaran *et al.* 2004). Each agent has multiple variables: one for the start time of each meeting it participates in, with 10 timeslots as values. Mutual exclusion constraints are imposed on the variables of an agent, and equality constraints are imposed on the corresponding variables of all agents

Agents	10	20	30	50	100
Meetings	3	9	11	19	39
Variables	10	31	38	66	136
Constraints	10	38	40	76	161
# of messages	35 / 9	778 / 30	448 / 37	3390 / 65	9886 / 135
Max message size	1 / 100	1 / 1000	1 / 100	1 / 1000	1 / 1000
Total Goods	35 / 360	778 / 2550	448/1360	3390 / 10100	9886 / 16920

Table 3: ODPOP vs DPOP tests on meeting scheduling (values are stated as ODPOP / DPOP)

involved in the same meeting. Private, unary constraints placed by an agent on its own variables show how much it values each meeting/start time. Random meetings are generated, each with a certain utility for each agent. The objective is to find the schedule that maximizes the overall utility.

Table 3 shows how our algorithm scales up with the size of the problems. All experiments are run on the FRODO multiagent simulation platform. The values are depicted as ODPOP / DPOP, and do not include the DFS and VALUE messages (identical). The number of messages refers to *ASK/GOOD* message pairs in $ODPOP$ and *UTIL* messages in $DPOP$. The maximal message size shows how many utilities are sent in the largest message in *DPOP*, and is always 1 in ODPOP (a single good sent at a time). The last row of the table shows significant savings in the number of utilities sent by ODPOP (*GOOD* messages) as compared to DPOP (total size of the *UTIL* messages).

Concluding Remarks

We proposed a new algorithm, ODPOP, which uses linear size messages by sending the utility of each tuple separately. Based on the best-first assumption, we use the principle of open optimization (Faltings & Macho-Gonzalez 2005) to incrementally propagate these messages even before the utilities of all input tuples have been received. This can be exploited to significantly reduce the amount of information that must be propagated. In fact, the optimal solution may be found without even examining all values of the variables, thus being possible to deal with unbounded domains.

Preliminary experiments on distributed meeting scheduling problems show that our approach gives good results when the problems have low induced width.

As the new algorithm is a variation of DPOP, we can apply to it the techniques for self-stabilization (Petcu & Faltings 2005c), approximations and anytime solutions (Petcu & Faltings 2005a), distributed implementation and incentive-compatibility (Petcu, Faltings, & Parkes 2006) that have been proposed for DPOP.

References

Collin, Z.; Dechter, R.; and Katz, S. 1991. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Faltings, B., and Macho-Gonzalez, S. 2005. Open constraint programming. *Artificial Intelligence* 161(1-2):181–208.

Freuder, E. C. 1985. A sufficient condition for backtrack-bounded search. *Journal of the ACM* 32(14):755–761.

Hamadi, Y.; Bessière, C.; and Quinqueton, J. 1998. Backtracking in distributed constraint networks. In *ECAI-98*, 219–223.

Kask, K.; Dechter, R.; and Larrosa, J. 2005. Unifying cluster-tree decompositions for automated reasoning in graphical models. *Artificial Intelligence*.

Maheswaran, R. T.; Tambe, M.; Bowring, E.; Pearce, J. P.; and Varakantham, P. 2004. Taking DCOP to the realworld: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS-04*.

Modi, P. J.; Shen, W.-M.; Tambe, M.; and Yokoo, M. 2005. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AI Journal* 161:149–180.

Petcu, A., and Faltings, B. 2005a. Approximations in distributed optimization. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'05)*.

Petcu, A., and Faltings, B. 2005b. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*.

Petcu, A., and Faltings, B. 2005c. Superstabilizing, fault-containing multiagent combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*.

Petcu, A.; Faltings, B.; and Parkes, D. 2006. MDPOP: Faithful Distributed Implementation of Efficient Social Choice Problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*.

Silaghi, M.-C.; Sam-Haroud, D.; and Faltings, B. 2000. Distributed asynchronous search with private constraints. In *Proc. of AA2000*, 177–178.

Solotorevsky, G.; Gudes, E.; and Meisels, A. 1996. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, 561–562.

Yokoo, M.; Durfee, E. H.; Ishida, T.; and Kuwabara, K. 1998. The distributed constraint satisfaction problem - formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering* 10(5):673–685.

Zivan, R., and Meisels, A. 2004. Concurrent Dynamic Backtracking for Distributed CSPs. In *Lecture Notes in Computer Science*, volume 3258. Springer Verlag. 782 – 787.