Journal of Information Processing Vol. 19 201–210 (May 2011)

Regular Paper

Efficient Representation of Constraints and Propagation of Variable–Value Symmetries in Distributed Constraint Reasoning

Xavier Olive[†1] and Hiroshi Nakashima[†1]

In this paper, we discuss variable and value symmetries in distributed constraint reasoning and efficient methods to represent and propagate them in distributed environments. Unlike commonly used centralised methods which detect symmetries according to their global definition, we suggest here to define them at the individual constraint level, then define operations on those symmetries in order to propagate them through the depth-first search tree that is generated in efficient distributed constraint reasoning algorithms. In our algorithm, we represent constraints (or utility functions) by a list of costs: while the usual representation lists one cost for one assignation, we drastically reduce the size of that list by keeping only one cost for one class of equivalence of assignations. In practice, for a constraint with n symmetric variables defined on a domain of n symmetric values, this approach cuts down the size of the list of costs from n^n to $p(n)$ (partition function of n), i.e., from 10^{10} to 42 when $n = 10$. We henceforth devised algorithms to process the sparse representations of utility functions and to propagate them along with symmetry information among distributed agents. We implemented this new representation of constraints and tested it with the DPOP algorithm on distributed graph colouring problems, rich in symmetries. Our evaluation shows that in 19% of execution instances we cut down 10 times the volume of communication spent, while no significant overhead appears in non symmetrical executions. These results open serious perspectives on a possible bounding of memory and communication bandwidth consumption in some subclass of distributed constraint reasoning problems.

1. Introduction

Distributed constraint optimisation is an efficient programming paradigm which expresses relations between variables in forms of constraints. Those variables are owned by a set of distributed agents, and only the agents involved in one constraint are aware of its existence. Keeping the definition of constraints distributed is particularly relevant when computational and communication power is limited, or when privacy concerns are raised by entities not willing to share sensitive information.

Symmetry processing in constraint reasoning is a research field in itself: symmetric problems being redundant in their definition, the literature is rich in methods offering to break symmetries in order to avoid revisiting equivalent states in search algorithms. Gent[1)] in particular published a comprehensive overview of centralised methods to deal with symmetric problems.

The distribution of data, however, is an obstacle to symmetry detection since the knowledge of the full definition of a problem is necessary to acknowledge their existence. We started in Refs. 2), 3) to consider the possibility of exploiting a weaker type of "partial" symmetries and of propagating them. Partial symmetries are applicable to a single constraint or to a subset of constraints of the whole problem.

In this paper, we present a way to exploit the existence of partial symmetries at the finest granularity level, i.e., inside individual constraints for an efficient representation. Then, we present a way to propagate those symmetries in order to preserve the efficiency and compactness of our simplified data structure.

Eventually, we challenge our new way to represent constraints in a distributed version of the well-known graph colouring problem, and show we manage to cut down the total volume of communication spent by the DPOP algorithm[4)] and to have significant improvement. We suggest that this result may have some serious impact provided we can identify a subclass of distributed constraint reasoning problems for which the exploitation of symmetries would bound the memory consumed.

2. Distributed Constraint Reasoning

Definition 1 (DCSP). *A distributed constraint satisfaction problem (DCSP) is a constraint satisfaction problem (CSP) where variables are distributed over different agents. It consists of a finite set of variables $x_1, \cdots, x_n$, a set of domains $d_1, \cdots, d_n$, a set of agents $a_1, \cdots, a_k$, and a set of constraints $c_1, \cdots, c_t$. Each variable is owned by a unique agent.*

†1 Kyoto University

We assume that if a constraint involves several variables, all agents owning any of these variables have a consistent definition of the constraint.

Definition 2 (Neighbourhood)**.** *Each constraint has a scope of variables, thus a scope of agents. Two variables are neighbours if they share at least a constraint. Similarly, if two variables are neighbours, then the agents they belong to are neighbours.*

Constraints fall into two categories: local (private) and global (distributed) constraints. Each agent owns a local subproblem, which is a partial view of the global problem. This global problem is the union of all the local subproblems.

2.1 Constraint Representation

Definition 3 (Assignation)**.** *An assignation is a tuple* $\{y_1 = u_1, \ldots, y_m = u_m\}$ *where* $u_i \in d_i$ *is a possible value for variable* y_i*.*

Definition 4 (Constraint, extensive representation)**.** *A constraint is an evaluation function (or utility function) defined on a scope of variables* $\{y_1, \ldots, y_m\}$*, associating a value to each possible assignation on the scope.*

There are two traditional ways to represent a constraint, an intensive and an extensive one. An extensive representation of a constraint lists all the variables, all their domains, and associates a constraint value (or *utility value*) to each assignation. It is a very expensive albeit simple way to represent a constraint: for k variables, each ranging on a domain of n values, the extensive representation will list all the possible n^k assignations.

On the other hand, an intensive representation is a very compact way to store a constraint: for example $\sum x_i = 1$, or $\texttt{alldifferent}(x_i)$, which represents the global constraint $x_j \neq x_k$ for all $j \neq k$. However, it can be very expensive to constantly evaluate its expression. Also, in constraint propagation algorithms, it is not convenient to intensively represent the junction of two constraints or to check their consistencies. As a consequence, the extensive representation is often more appropriate to CSP and DCSP algorithms.

2.2 DFS Tree

Yokoo presented the first distributed Asynchronous Backtracking [5] to solve distributed constraint satisfaction problems. The main issue it raised together with other pioneer algorithms was the fact it kept increasing the size of their variables' neighbourhoods during the resolution process. ADOPT [6] and DPOP [4] family's algorithms started to group variables according to their initial neighbourhoods, i.e., according to the constraints they share, using a DFS tree obtained from a depth-first search (DFS) of the *constraint graph* of the problem with variables as nodes and constraints as edges as shown in the left half of **Fig. 1**.

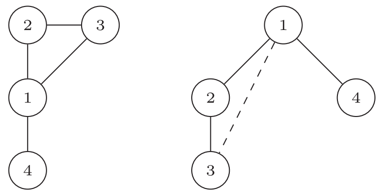


Fig. 1 A constraint graph and its DFS tree.

Definition 5 (DFS tree)**.** *A DFS tree is a rooted and directed spanning pseudo-tree of the constraint graph such that any two neighbours in the original graph are both in the same branch.*

Note that Definition 5 assures that all neighbouring pairs of variables in a constraint graph are involved in an ancestor/descendent relationship in the corresponding DFS tree (see $(1,2)$, $(1,3)$, $(1,4)$ and $(2,3)$ couples in Fig. 1.)

Also, as the edges of the DFS tree only represent some of the constraints, we introduce the concept of *back edge* to the DFS tree in order to denote the existence of a constraint between two variables which are not connected by a simple edge. A variable connected to an ancestor via a back edge is called a *pseudo-child* and the ancestor is called a *pseudo-parent.* In the figure, the dashed line is a back edge, and 1 and 3 are resp. pseudo-parent and pseudo-child of each other.

Definition 6 (Separator)**.** *Each variable x has a separator $sep(x)$, defined as the set of its ancestors connected with an edge or a back edge to x or to any descendent of x. More specifically,*

$$sep(x) = ancestors(x) \bigcap \left(\bigcup_{y \in \{x\} \cup descendents(x)} pseudoparents(y) \right)$$

In other words, the separator of a variable x is the set of ancestor variables of x constrained with x or with any descendant of x. If the DFS tree is a real tree, all the separators do not have more than one element.

2.3 DPOP Algorithm

ADOPT[6] was the first distributed search algorithm using DFS tree structures. Later came DPOP[4], which performs a distributed version of the bucket elimination algorithm[7]. After creating a DFS tree, i.e., labelling each variable as parent/child and pseudo-parent/pseudo-child of its neighbours, it proceeds in two phases: a `UTIL` propagation, aggregating constraints from the leaves to the root of the tree, followed by a top-down `VALUE` propagation in order to find optimal assignations.

Definition 7 (`UTIL` message). *The `UTIL` message sent from a variable v to its parent is a multidimensional matrix representing the global constraint linking the variables in the separator of v. The matrix has n dimensions, one per variable in the separator of $v = \{y_1, \cdots, y_n\}$, and is filled with the costs $f(u_1, \cdots, u_n)$ of each assignation $\{y_1 = u_1, \cdots, y_n = u_n\}$.*

The `UTIL` propagation phase starts from the leaves and goes up to the root. Each node aggregates and optimises constraints, namely joins and projects `UTIL` messages coming from its children and sends to its parent a representation of relations with its ancestors via a new `UTIL` message.

Definition 8 (Junction). *Let $f_1, \cdots, f_n$ be utility functions defined on scopes $S_1, \ldots, S_n$. The junction $\sum f_i$ is defined on the union of the scopes $S = \bigcup S_i = \{y_1, \cdots, y_k\}$. For each possible assignation in S, we have $(\sum f_i)(u_1, \ldots, u_k) = \sum(f'_i(u_1, \ldots, u_k))$, where $f'_i(S) = f_i(S_i)$ for all assignations in S.*

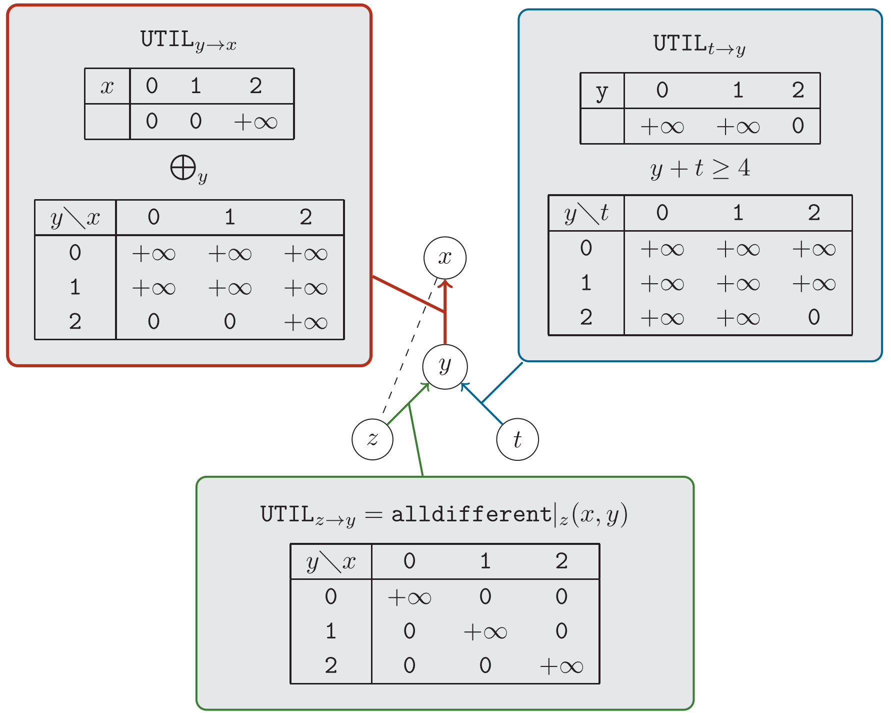
The junction is an aggregation operation which a node applies on the resulting constraints, or utility functions, coming from its different subtrees. If variable x, linked with its ancestors by constraint f_0, has k children $y_1, \cdots, y_k$, it receives from each of them a `UTIL` message with cost functions $f_1, \cdots, f_k$ of respectively $sep(y_1), \cdots, sep(y_k)$. The node for variable x then sums up all these constraints in $(f_0 + f_1 + \cdots + f_k)$ defined on $parents(x) \cup sep(y_1) \cup \cdots \cup sep(y_k) = sep(x) \cup \{x\}$.

Definition 9 (Projection). *Let f be a function of a set of variables $s = \{x, y_1, \cdots, y_n\}$. We define the projection of f on x as the function $f|_x$ of $s - \{x\}$ such that for every assignation $\{y_1 = u_1, \cdots, y_n = u_n\}$ of $s - \{x\}$, $f|_x(u_1, \cdots, u_n) = \min_x f(x, u_1, \cdots, u_n)$.*

The projection is an optimisation operation where a node picks the optimal assignation for the variable it holds for all possible assignations of the variables in its separator. With the same notation as above, the node for x will project the function $(f_0 + f_1 + \cdots + f_k)$ and create $f|_x$ of $sep(x)$ to be sent to its parent. For a leaf node, x will be projected out of only f_0.

For a root node holding variable r, f_0 does not exist. The `UTIL` messages received contain only functions of r: projecting it out of the junction determines its optimum value r_0. This optimum value is sent by the root to its children through `VALUE` messages. When a node receives such a message, it chooses its optimum according to the values received, creates new `VALUE` messages with its optimum and send it to its children until the leaves.

Consider the constraint reasoning problem consisting of four variables x, y, z, t each defined on $\{0, 1, 2\}$ and constrained by $\texttt{alldifferent}(x, y, z)$ and $y + t \geq 4$.



$\texttt{UTIL}_{y\to x}$

x	0	1	2
	0	0	$+\infty$

$\bigoplus_y$

$y \setminus x$	0	1	2
0	$+\infty$	$+\infty$	$+\infty$
1	$+\infty$	$+\infty$	$+\infty$
2	0	0	$+\infty$

$\texttt{UTIL}_{t\to y}$

y	0	1	2
	$+\infty$	$+\infty$	0

$y + t \geq 4$

$y \setminus t$	0	1	2
0	$+\infty$	$+\infty$	$+\infty$
1	$+\infty$	$+\infty$	$+\infty$
2	$+\infty$	$+\infty$	0

$\texttt{UTIL}_{z\to y} = \texttt{alldifferent}|_z(x, y)$

$y \setminus x$	0	1	2
0	$+\infty$	0	0
1	0	$+\infty$	0
2	0	0	$+\infty$

Fig. 2 The DPOP algorithm.

Figure 2 displays a possible DFS tree. The `alldifferent` constraint is a $3\times3\times3$ table filled with $+\infty$ except for the following assignations where the utility values would be 0: $(0,1,2)$, $(0,2,1)$, $(1,0,2)$, $(1,2,0)$, $(2,0,1)$ and $(2,1,0)$. Similarly, the constraint linking y and t by $y+t \geq 4$ is explained in Fig. 2. After the projections of z and t, y receives both $\texttt{UTIL}_{z\rightarrow y}$ and $\texttt{UTIL}_{t\rightarrow y}$, before joining them into $\bigoplus_y$. Then y is projected out of $\bigoplus_y$ and $\texttt{UTIL}_{y\rightarrow x}$ is sent to the root node x. x can then choose an assignation, for example 1 (0 is also possible), and propagate it down: y has to be equal to 2, then z should be 0 whilst t must be equal to 2.

DPOP's advantage lies in the number of messages sent. Once the DFS tree is generated, it sends no more than twice as many messages as the number of edges in the DFS tree. However, its weakness lies in the size of those messages: the size of the biggest message grows exponentially with the size of the biggest separator of the tree, also called *induced width*. That is, if a separator has n variables over domains $d_1, \ldots, d_n$, the size of the corresponding `UTIL` message is proportional to $\prod |d_i|$. This paper proposes an innovative method to attack this bottleneck.

3. Symmetry in Constraint Reasoning

Exploitation and breaking of symmetry in constraint reasoning have been a research field in itself. As symmetries are omnipresent in nature, using them in various fields of physics or engineering can make problems easier and faster to solve. In constraint programming, symmetry breaking 8),9) can avoid revisiting equivalent states. The knowledge of symmetry itself can also let the software induce new constraints 10) that would make the resolution process faster.

3.1 Definition

Gent presents in Ref. 1) various equivalent definitions of symmetry that are often used in the constraint programming literature. We present here only two definitions that are convenient for our demonstration.

Definition 10 (Symmetry)**.** *A symmetry over a CSP is a permutation on variables and values that leaves the whole set of constraints unchanged.*

For example, if (x_1, y_1, x_2, y_2, z) are all defined on $\{0,1\}$ and constrained by $x_i+2y_i = z$, $\sigma = \{(x_1 \rightleftharpoons x_2), (y_1 \rightleftharpoons y_2)\}$ is a symmetry: indeed, as $\sigma(\{x_1+2y_1 = z\}) = \{x_2+2y_2 = z\}$ and $\sigma(\{x_2+2y_2 = z\}) = \{x_1+2y_1 = z\}$, σ leaves the set of constraints globally unchanged. Therefore, we can reduce the search effort by solving only $x_1+2y_1 = z$ and applying the symmetry to the solution in order to find the solution of the whole original problem.

We can also reformulate the definition in more mathematical terms.

Definition 11 (Symmetry)**.** *A symmetry σ over a CSP is an automorphism on the set of assignations that leaves their utility values unchanged. In other words, $\sigma : d_1 \times \cdots \times d_n \longrightarrow d_1 \times \cdots \times d_n$; then, for all $\alpha \in d_1 \times \cdots \times d_n$, the equality $c(\alpha) = c(\sigma \cdot \alpha)$ stands (where $c(\alpha)$ is the global utility of assignation α, sum of all the constraints of the whole problem.)*

A prerequisite to both definitions is that we can define the variable permutation $x_i \rightleftharpoons x_j$ iff $d_i = d_j$ and the value permutation $u_i \rightleftharpoons u_j$ iff for all k, $u_i \in d_k \Leftrightarrow u_j \in d_k$. The detection of symmetries over centralised CSPs has been studied in Ref. 11) with the help of group theory. The definition of orbit, inherent to the study of groups, will be of some help in upcoming sections.

Definition 12 (Orbit)**.** *The orbit of an element α is the set of elements which are images of α through any symmetry σ in the group G, and is noted $G \cdot \alpha = \{\sigma \cdot \alpha \,|\, \sigma \in G\}$.*

Proposition 1. *We can partition the set of assignations into orbits.*

Proof. If the symmetry group is empty, then there are as many orbits as assignations. By taking assignations in order, we can merge the orbits of each assignation α and of its image $\sigma \cdot \alpha$. □

Definition 13 (Generator)**.** *We name generator of an orbit $G \cdot \alpha$ the element $\dot{\alpha}$ which is the first in lexicographical order among all elements in $G \cdot \alpha$. In other words,*

$\forall \alpha \in G \cdot \dot{\alpha}$, we have $\dot{\alpha} \ll \alpha$

We name γ the function associating $\dot{\alpha}$ to α.

Computing the γ function for assignations with all interchangeable variables and all interchangeable values, proceeds as follow. We illustrate our point with example assignation $(2,0,2)$ and domain $\{0,1,2\}$:

- first we count the occurrences of each values, and swap the most used value with the first value in lexicographical order, then the second most used with the second value, etc. $(2,0,2)$ becomes $(0,1,0)$ through a combination of value permutations;

- then we sort the result in lexicographical order: $(0, 1, 0)$ becomes $(0, 0, 1)$ through a combination of variable permutations.

In conclusion, $\gamma(2, 0, 2) = (0, 0, 1)$. For the general case, the reader can refer to Ref. 12), which also shows an efficient algorithm to list all orbits from a symmetry group. This orbit listing procedure coincides with the partition function on the simplest case of fully symmetric values and variables.

3.2 Symmetries in Distributed Context

Numerous publications presented how to exploit detected symmetries, e.g., in search algorithms by dominance detection. In a distributed context however, the symmetry detection is compromised by the distribution of data: there is no easy way to check that a permutation leaves the whole set of constraints unchanged, since all agents have only a partial view over the problem. Henceforth, the distribution of the definition requires a different approach.

We raised those problems in Ref. 2) and introduced the notion of partial symmetry, then stated as follows a condition for a partial symmetry to be also a symmetry on the global definition of the problem.

Definition 14 (Partial representation)**.** *A partial representation of a problem p is the restriction of p to a subset of variables $\mathcal{V}$, the neighbours of the variables in $\mathcal{V}$, and the constraints involving any variable in $\mathcal{V}$. Each agent naturally owns a partial representation of the problem restricted to its variables.*

Definition 15 (Partial symmetry)**.** *A partial symmetry over a CSP is a symmetry over a partial representation of the CSP. The symmetries detected by an agent which has a partial view over the problem are partial symmetries.*

Theorem 2. *If σ is a partial symmetry for all agents owning a variable in σ and for all their neighbouring agents, σ is a symmetry of the whole problem.*

Theorem 2 (proved in Ref. 2)) implies that distributed agents can cooperate and detect a global symmetry from their own local symmetries by finding a subset of those local symmetries which they all agree on. However, this approach fails to detect the global symmetries which are not local ones.

The SymDPOP algorithm[3)] breaks through by analysing the DFS tree from the leaves and propagating symmetries up to the root conjointly with constraints expressed as `UTIL` messages. Indeed, if two `UTIL` messages have interchangeable separators, only one of those messages can be sent. Its strength lies in the fact it does not require that all the agents agree on a symmetry to propagate it: only the source and destination agents have to. Although this approach significantly cuts the number and volume of messages sent on both symmetrical and non symmetrical problems, analyzing only the permutability of separators limits its impact in so far as the only symmetries found are variable symmetries.

Subsequently, we reuse in this paper the idea of propagating symmetries together with constraints, but generalise it in order to deal with all kind of symmetries. We manage to have a more compact representation of constraints, hence reduce the volume of data used.

3.3 Constraint Representation

The main problem of the extensive representation of constraints as stated in Section 2.1 is the immense amount of data consumed, as it is exponential in the number of variables in the constraint's scope: in DPOP, a regular constraint is stored in an `Hypercube`, containing the list of variables, the list of their domains, and the list of utilities associated with every possible assignation. If we consider the Definition 12 of orbits together with Proposition 1, we can significantly simplify the representation of a constraint.

Indeed, once we know a group of symmetries valid for one specific constraint, we can partition its set of assignations into orbits. All the elements of a single orbit do have the same constraint valuation (or the same utility value) so we can store only one utility per orbit. We name this structure `SparseHypercube` because of the analogy with sparse matrices.

For example, the constraint `alldifferent`(x, y, z) with all variables defined on $\{0, 1, 2\}$ is subject to a set of variable symmetries – all variables are interchangeable – and of value symmetries – all values are interchangeable. As shown on **Fig. 3**, we can cut the number of utility values stored from $3^3 = 27$ to 3. Here, the intrinsic meaning of each orbit is "all elements are the same" (orbit of $(0, \dot{0}, 0)$), "2 elements are identical, 1 is different" (orbit of $(0, \dot{0}, 1)$) and "all elements are different" (orbit of $(0, \dot{1}, 2)$). Note that in that case, when the size of domains n grows, the extensive representation of constraints will have n^3 elements whilst there will still be only 3 orbits. More generally, the number of elements for k symmetric variables and $n \geq k$ symmetric values are n^k in the extensive representation but $p(k)$ in our intensive one where $p(k)$ is the partition

- $x, y, z \in \{0,1,2\}$
- $(0,0,0) \to +\infty$,
  $(0,0,1) \to +\infty$,
  $\dots$
  $(0,1,1) \to +\infty$
  $(0,1,2) \to 0$,
  $\dots$
  $(2,2,2) \to +\infty$

- $x, y, z \in \{0,1,2\}$
- $G = x \rightleftharpoons y \rightleftharpoons z$ and $0 \rightleftharpoons 1 \rightleftharpoons 2$
- $G \cdot (0,\dot{0},0) \to +\infty$,
  $G \cdot (0,\dot{0},1) \to +\infty$,
  $G \cdot (0,\dot{1},2) \to 0$

Fig. 3 Extensive and orbits representation of an `alldifferent` constraint.

- $x, y \in \{0,1,2\}$
- $G = x \rightleftharpoons y$

- $G \cdot (0\dot{,}0) \to 0$,
  $G \cdot (0\dot{,}1) \to 1$,
  $G \cdot (0\dot{,}2) \to 1$,
  $G \cdot (1\dot{,}1) \to 1$,
  $G \cdot (1\dot{,}2) \to +\infty$,
  $G \cdot (2\dot{,}2) \to +\infty$

Fig. 4 Representation of constraint c with orbits.

function of k.

For another example, we can consider the constraint c on variables x and y, both defined on $\{0,1,2\}$, and assigning utility value 0 if their sum is equal to 0, 1 if their sum is equal to 1 or 2 and $+\infty$ otherwise. Variables x and y are permutable so we can represent the constraint as in **Fig. 4** with 6 utility values instead of 9.

3.4 Symmetry Propagation

The idea of summarising all the assignations into orbits is beneficial in algorithms provided we can propagate this concept. In DPOP for example, where constraints, represented as `UTIL` messages are joined and propagated up the DFS tree, we have to make sure that we can still group assignations into orbits after operating DPOP basic operations, namely projection and junction.

3.4.1 Projection

During the projection operation, as one variable is taken out of the scope of the constraint, it must also be taken out of G.

Definition 16 (Projection)**.** *Let G be a group of symmetries on the scope of variables $(x, y_1, \cdots, y_n)$. We name $G|_x$, the group of symmetries included in G which do not involve any permutation with the variable x.*

- $x, y, z \in \{0,1,2\}$
- $G = x \rightleftharpoons y \rightleftharpoons z$ and $0 \rightleftharpoons 1 \rightleftharpoons 2$
- $G \cdot (0,0,0) \to +\infty$,
  $G \cdot (0,0,1) \to +\infty$,
  $G \cdot (0,1,2) \to 0$

- $y, z \in \{0,1,2\}$
- $G = y \rightleftharpoons z$ and $0 \rightleftharpoons 1 \rightleftharpoons 2$
- $G \cdot (0,0) \to +\infty$,
  $G \cdot (0,1) \to 0$

Fig. 5 `alldifferent` constraint and its projection along x.

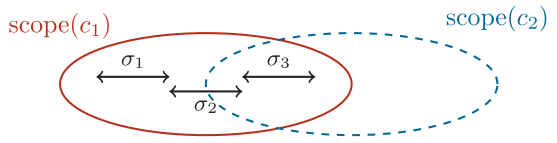


Fig. 6 Definition of symmetry group junction.

Proposition 3. *If G is a symmetry group for the function f defined on the scope $(x, y_n, \cdots, y_n)$, then $G|_x$ is a symmetry group for the projection of f on x.*

Proof. Let σ_x be a symmetry in $G|_x$, we have

$$\begin{aligned} f|_x\left(\sigma_x \cdot (u_1, \cdots,\ u_n)\right) &= \min_x f(x, \sigma_x \cdot (u_1, \cdots, u_n)) \\ &= \min_x f(x, u_1, \cdots, u_n) \\ &= f|_x(u_1, \cdots, u_n) \end{aligned}$$

□

Figure 5 shows the result of the projection of a variable out of an `alldifferent` constraint. For the sake of clarity, we detail here how to compute $f|_x(0,1)$.

$$\begin{aligned} f|_x(0,1) &= \min\{f(0,0,1), f(1,0,1), f(2,0,1)\} \\ &= \min\{f\cdot\gamma(0,0,1), f\cdot\gamma(1,0,1), f\cdot\gamma(2,0,1)\} \\ &= \min\left\{f(0,\dot{0},1), f(0,\dot{0},1), f(0,\dot{1},2)\right\} \\ &= \min\{+\infty, +\infty, 0\} = 0 \end{aligned}$$

3.4.2 Junction

Figure 6 helps understanding the following definition, describing sufficient conditions for a symmetry on a constraint c_1 to be also a symmetry on the junction of c_1 with another constraint c_2.

Definition 17 (Junction). *Let G_1 (resp. G_2) be a group of symmetries on the constraint c_1 (resp. c_2) defined on the scope of variables S_1 (resp. S_2), we define the junction of those groups $G_1 \oplus G_2$ by considering the elements of G_1 (resp. G_2) in order:*

- *if $\sigma_1 \in G_1$ permutates variables in $S_1 \cap \overline{S_2}$, then $\sigma_1 \in G_1 \oplus G_2$,*
- *if $\sigma_2 \in G_1$ permutates $x \in S_1 \cap \overline{S_2}$ with $y \in S_1 \cap S_2$, then $\sigma_2 \notin G_1 \oplus G_2$,*
- *if $\sigma_3 \in G_1$ permutates variables $y \in S_1 \cap S_2$, then $\sigma_3 \in G_1 \oplus G_2$ iff. $\sigma_3 \in G_2$.*

Proposition 4. *If G_i is a symmetry group for function f_i, then $\bigoplus G_i$ is a symmetry group for function $\sum f_i$.*

Proof. (for 2 functions) Let σ be a symmetry in $G_1 \oplus G_2$ and α be an assignation in the scope of $(f_1 + f_2)$, we have

$$\begin{aligned}(f_1 + f_2)(\sigma \cdot \alpha) &= f_1(\sigma \cdot \alpha) + f_2(\sigma \cdot \alpha)\\ &= f_1(\alpha) + f_2(\alpha)\\ &= (f_1 + f_2)(\alpha)\end{aligned}$$

as the equality $f_i(\sigma \cdot \alpha) = f_i(\alpha)$ is guaranteed by the definition of $G_1 \oplus G_2$, considering case 1 and case 3 for $\sigma \in G_1$ and for $\sigma \in G_2$. The proposition for the cases of three or more functions can be easily proved by a simple induction. □

Proposition 4 gives us the means to build $\sum f_i$ for all generators $\dot{\alpha}$ of orbits of $\bigoplus G_i$ by summing up all $f_i(\gamma_i(\dot{\alpha}))$ where γ_i is the generator function with respect to G_i. For example, **Fig. 7** shows the result of the junction of one `alldifferent` constraint with one `oneeven` constraint (one of the variables in the scope of `oneeven` must be equal to 0 or 2). After listing all possible orbits for $G_1 \oplus G_2$, we compute $(f_1 + f_2)(x)$ for all $\alpha \in \gamma(G_1 \oplus G_2)$. For the sake of clarity, we detail here how to compute $(f_1 + f_2)(0, 1, 0, 1)$.

$$\begin{aligned}(f_1 + f_2)(0,1,0,1) &= f_1(0,1,0) + f_2(0,1,1)\\ &= f_1 \cdot \gamma_1(0,1,0) + f_2 \cdot \gamma_2(0,1,1)\\ &= f_1(0,\dot{0},1) + f_2(0,\dot{0},1) = +\infty\end{aligned}$$

4. Performance

In order to evaluate this different way of representing constraints, we chose to compare the volume of data used by the DPOP algorithm on distributed graph colouring problems.

- $x, y, z \in \{0, 1, 2\}$
- $G_1 = x \rightleftharpoons y \rightleftharpoons z$ and $0 \rightleftharpoons 1 \rightleftharpoons 2$
- $G_1 \cdot (0,0,0) \to +\infty$, $G_1 \cdot (0,0,1) \to +\infty$, $G_1 \cdot (0,1,2) \to 0$

- $x, y, t \in \{0, 1, 2\}$
- $G_2 = x \rightleftharpoons y \rightleftharpoons t$ and $0 \rightleftharpoons 2$
- $G_2 \cdot (0,0,0) \to 0$, $G_2 \cdot (1,1,1) \to +\infty$, $G_2 \cdot (0,0,1) \to 0$, $G_2 \cdot (0,0,2) \to 0$, $G_2 \cdot (1,1,0) \to 0$, $G_2 \cdot (0,2,1) \to 0$

- $x, y, z, t \in \{0, 1, 2\}$
- $G = x \rightleftharpoons y$ and $0 \rightleftharpoons 2$
- 18 orbits (for 81 assignations)

Fig. 7 `alldifferent` (left) constraint and `oneeven` (right) constraint and their junction (down).

4.1 Distributed Graph Colouring

The distributed graph colouring problem is a distributed version of the well-known problem which consists of assigning colours to nodes of a graph, making sure that any two neighbouring nodes get a different colour assigned.

Each node x_0 has a constraint stating the impossibility of having identical colours for this node and each of its neighbours (x_i). We modelled that constraint as a single constraint `graphcolouring`$(x_0, x_1, \ldots x_{k_0})$ (k_0 being the number of neighbours of x_0). The utility function assigns $+\infty$ if there exists an i such as $x_0 = x_i$, and 0 otherwise. We also have a group of value symmetries (all colours are permutable) and of variable symmetries (all x_i with $i \neq 0$ are permutable).

The distributed graph colouring problem is particularly relevant in so far as it is equivalent to many real life problems, such as the assignation of channels to wireless routers.

4.2 Results

We generated several distributed graph colouring problems with 10 colours, 3 agents, each owning 10 variables. For each instance of the problem, we measured

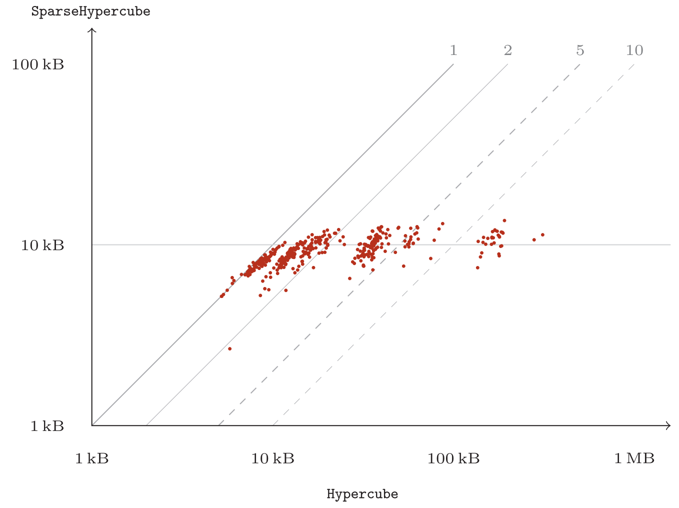


Fig. 8 Data volume used for each of 400 executions.

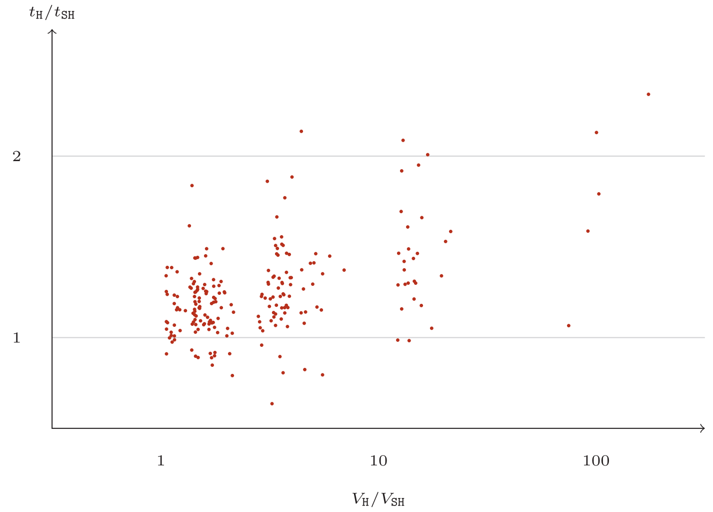


Fig. 9 Execution speed-up.

the total volume of data exchanged, which is the widely acknowledged bottleneck for the DPOP algorithm.

On **Fig. 8**, we compared data volumes with regular extensive Hypercubes and with condensed SparseHypercubes, which take symmetries into account; we drew for each of those 400 executions a dot, whose coordinates are the total volume of data exchanged with Hypercubes on the abscissa and with SparseHypercubes on the ordinate. All the points on the right side of each of the diagonal lines are executions for which the total volume of information sent with SparseHypercube is n times smaller than with Hypercube, n being the number written on the top right-hand side of this line. We find that in 68% of executions, by using symmetry information in constraint representation and by propagating those information, we cut the total volume of data by more than 2; then by more than 5 in 30% of executions, and by more than 10 in 19% of executions. More interestingly, the total amount volume of data seems to be bounded to about 10 kB when we use SparseHypercube structures. Also, there seem to be clusters of data volumes when we use Hypercube structures.

Depending on some characteristics of the problem (not only the number of variables, the size of constraints, the density of the constraint graph), it appears that using SparseHypercube cuts the scarce repartition of volume of data used to the better case. Considering the way partial symmetries are propagated along the whole problem could be a first step forward in order to find a subclass of distributed constraint programming problems for which exploitation of symmetries would have a strong bounding effect on the volume of constraints used.

Figure 9 displays the execution time speed-up in function of the data volume difference (V stands for "data volume", t for "execution time", H for Hypercube and SH for SparseHypercube). The time has been measured on T2K supercomputer[13], running FRODO[14] software with mpj-express[15] based communication structures on 4 Fujitsu HX600 nodes[⋆1]. We observe a significant speed-up of 30% in average. The negative speed-up for instances with few symmetries is due to few fine optimisations in regular DPOP that we had to discard. In reasonably symmetrical instances, the time spent to compute the γ function[12] during projection and junction operations is largely compensated by the economy we make

⋆1 Each node consists of 4 quad-core Opteron 8356, 32 GB RAM, java-6-sun-1.6.

in data volume in the communication process. We can also legitimately infer that in a real-life distributed context with a lower bandwidth than MPI, the speedup would get even more significant.

Note also that solving some instances of the same problem with 15 colours with `Hypercube` is often impossible with 256 MB of memory because of exponential explosion. On the other hand, `SparseHypercube` breaks colour symmetries and solves the problem with an unlimited number of colours.

5. Related Works

Symmetry treatment is an important research topic in several areas of computer science, including model-checking[16)] and constraint programming. The question has been addressed from several angles in centralised constraint programming: static[9)] or dynamic symmetry breaking, problem reformulation, symmetry definitions[8)] and representations, detection of symmetries[11)]; Ref. 1) presents a comprehensive overview of various methods. We first tackled the paradox between symmetry and distribution in Ref. 2) and the idea of propagating symmetry information appeared with our SymDPOP[3)] algorithm.

On top of these pieces, including our own unique work in distributed context, we made essential contributions to distributed constraint solving problems. While SymDPOP takes care of a particular case of variable symmetries, we generalised the idea of symmetry propagation and incorporated all variable/value symmetries in DPOP algorithms, expanding the range of application of SymDPOP. Not only did we reduce the number of `UTIL` messages, we also reduced their sizes by a representation based on classes of equivalences, or orbits. Eventually, we formulated and implemented the fundamental functions of DPOP algorithm, resp. junction and projection, to fit the orbit-based representation, hence reducing both memory space and computation cost.

6. Conclusions

We presented in this paper a method which utilises the symmetry properties of constraints in order to significantly cut the total communication volume spent during the DPOP resolution process. This approach uses the symmetry groups properties and represents classes of equivalences of assignations (orbits) instead of assignations. Symmetries are input locally at the level of constraints and propagated all along the DFS tree through junction and projection operations. This method contrasts with symmetry detection methods commonly used in a centralised context which consider symmetries on the global definition of problems. Rather, it takes advantage of partial symmetries which are valid only on subparts of the problem, propagates this information to neighbours, and finally cuts down 10 times the total volume of communication spent. This work opens perspectives on characterising distributed constraint programming problems for which the exploitation of symmetry could bound the exponential consumption of memory.

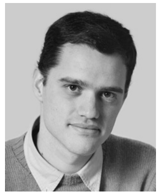
Acknowledgments The authors are deeply indebted to the LIA laboratory of EPFL, in particular Radoslaw Szymanek and Thomas Léauté for their FRODO software and their precious advice.

References

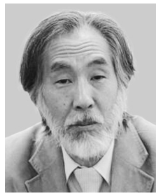
1) Gent, I., Petrie, K. and Puget, J.-F.: Symmetry in Constraint Programming, *Handbook of Constraint Programming*, Rossi, F., van Beek, P. and Walsh, T. (Eds.), Chapter 10, pp.329–376, Elsevier Science Ltd. (2006).
2) Olive, X. and Nakashima, H.: Breaking Symmetries in Distributed Constraint Programming Problems, *Proc. 11th International Workshop on Distributed Constraint Programming*, Hirayama, K., Yeoh, W. and Zivan, R. (Eds.) (2009).
3) Olive, X. and Nakashima, H.: SymDPOP: Adapting DPOP to exploit partial symmetries, *Proc. 12th International Workshop on Distributed Constraint Programming*, Lass, R. and Sultanik, E. (Eds.) (2010).
4) Petcu, A. and Faltings, B.: A Scalable Method for Multiagent Constraint Optimization, *Proc. 19th International Joint Conference on Artificial Intelligence*, Vol.19, p.266 (2005).
5) Yokoo, M. and Hirayama, K.: Algorithms for Distributed Constraint Satisfaction: A Review, *Autonomous Agents and Multi-agents Systems*, Vol.3, No.2, pp.198–212 (2000).
6) Modi, P., Shen, W., Tambe, M. and Yokoo, M.: ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees, *Artificial Intelligence*, Vol.161, No.1-2, pp.149–180 (2005).
7) Dechter, R.: Bucket elimination: A unifying framework for reasoning, *Artificial Intelligence*, Vol.113, No.1-2, pp.41–85 (1999).
8) Cohen, D., Jeavons, P., Jefferson, C., Petrie, K. and Smith, B.: Symmetry Definitions for Constraint Satisfaction Problems, *Constraints*, Vol.11, No.2, pp.115–137

(2006).
9) Roy, P. and Pachet, F.: Using Symmetry of Global Constraints to Speed up the Resolution of Constraint Satisfaction Problems, *Workshop on Non Binary Constraints, ECAI*, Vol.98 (1998).
10) Fahle, T., Schamberger, S. and Sellmann, M.: Symmetry Breaking, *Principles and Practice of Constraint Programming*, Walsh, T. (Ed.), LNCS 2239, pp.93–107, Springer (2001).
11) Puget, J.-F.: Automatic Detection of Variable and Value Symmetries, *Principles and Practice of Constraint Programming*, van Beek, P. (Ed.), LNCS 3709, pp.475–489, Springer (2005).
12) Olive, X. and Nakashima, H.: Around the Constructive Orbit Problem in Distributed Constraint Programming, *IPSJ SIG Notes*, Vol.2011-AL-134, No.22 (2011).
13) Nakashima, H.: T2K Open Supercomputer: Inter-University and Inter-Disciplinary Collaboration on the New Generation Supercomputer, *Internation Conference on Informatics Education and Research for Knowledge-Circulating Society* (2008).
14) Leauté, T., Ottens, B. and Szymanek, R.: FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization (2009). http://liawww.epfl.ch/frodo/
15) Baker, M., Carpenter, B. and Shafi, A.: MPJ Express: Towards thread safe Java HPC, *Proc. 2006 IEEE International Conference on Cluster Computing*, pp.1–10 (2006).
16) Clarke, E., Emerson, E., Jha, S. and Sistla, A.: Symmetry reductions in model checking, *Computer Aided Verification*, Springer, pp.147–158 (1998).

(Received July 31, 2010)
(Accepted February 4, 2011)
(Released May 11, 2011)



Xavier Olive was born in 1985. He received an Engineer degree in System Engineering and Information Technology together with a Master degree in Software Safety and Calculating Capacity from the French National Superior Engineering School of Aeronautics and Space (SUPAERO) in 2006. He received his Ph.D. degree from Kyoto University in 2011.



Hiroshi Nakashima was born in 1956. He received his M.E. and Ph.D. degrees from Kyoto University in 1981 and 1991, respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a Professor at Toyohashi University of Technology in 1997, and a Professor at Kyoto University in 2006. His current research interests are the architecture of parallel processing systems and the implementation of programming languages. He received the Motooka Award in 1988 and the Sakai Award in 1993. He is a Fellow of IPSJ, and a member of IEEE-CS, ACM, ALP and TUG.