SymDPOP: Adapting DPOP to exploit partial symmetries

Xavier Olive and Hiroshi Nakashima

Graduate School of Informatics, Kyoto University

Abstract. This work proposes a new approach to dealing with symmetries in Distributed Constraint Optimisation Problems by adapting the DPOP algorithm [1]. In contrast to an already proposed distributed preprocessing method leading to a problem redefinition [2], we present here a method that exploits the structure of DFS trees, with no explicit redefinition of the problem.
We exhibit the flexibility in the symmetry detection that this algorithm offers, then compare its performance with DPOP and the aforementioned preprocessing method. We demonstrate that SymDPOP significantly cuts down the total volume of communication spent on symmetric and partially symmetric problems, the latter being problems containing symmetric subproblems. We also stress the fact that SymDPOP minimises the overhead of symmetry detection by keeping the total volume of communication below that of DPOP.

Keywords. distributed constraint optimisation, symmetry breaking

1 Introduction

Distributed Constraint Optimisation is an efficient framework suitable for modelling naturally distributed problems. Those consist in different agents cooperating to solve an optimisation problem. Only a subset of agents knows about each constraint, which leads to the idea of natural distribution. Keeping the definition of a problem distributed may be relevant when computational and communication power is limited, or when privacy concerns are raised.

After Asynchronous Backtracking by Yokoo [3], several algorithms have been proposed to solve Distributed Constraint Satisfaction Problems (DCSP), including ADOPT [4] or DPOP [1]. As symmetry processing can be successful in traditional constraint programming [5, 6], an attempt [2] has been made to exploit them in a distributed context as a preprocessing step to any solving technique. Yet, this procedure requires that all the agents agree on how to reformulate the symmetric problem into a smaller one.

We present here SymDPOP, a new method for exploiting symmetries which is specific to DPOP algorithm and does not require any preprocessing step. We show that by adding extra information to the messages used by DPOP, we can reduce their number, their volume, as well as the workload of each agent.

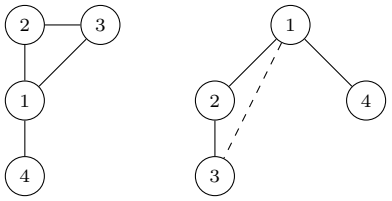

Fig. 1. A constraint graph and its DFS tree

We also show that unlike the preprocessed DPOP presented in [2], SymDPOP cuts down on the total volume (rather than number) of messages sent. Moreover, the work and communication saved in the symmetry detection process are not lost even if the detection process cannot find any symmetry.

2 Preliminaries

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 $a_1 \cdots a_n$ not necessarily all different, and a set of constraints $c_1 \cdots c_t$.*

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

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 DFS trees

Whereas the first algorithms designed to solve DCSP kept increasing the size of the neighbourhoods of their variables, a more reasonable approach used in DPOP or ADOPT groups variables according to the constraints they share.

Definition 3 (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.*

From a constraint graph, with variables as nodes and constraint as edges, as shown in Figure 1, we build a DFS tree. The edges of the DFS tree only represent some of the constraints. Thus, a *back edge* is an edge present in the constraint graph as well as in the DFS pseudo-tree but not in the tree part of the DFS tree. An agent 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 4 (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.*

Definition 5 (Induced width). *The induced width of a graph is the size of the largest separator of the nodes in the generated DFS tree.*

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. DPOP bounds the number of sent messages. However its weakness lies in the size of those messages which grows exponentially with the induced width of the DFS tree.

In order to generate a DFS tree in a distribute manner [7], all the agents label their variables as *non-visited*. Then, using a variable election algorithm, one of the agents is designated as root. Choosing the variable with the biggest neighbourhood as root favours well-balanced DFS trees, as the most linked variables are more likely to be around the centre of the constraint graph.

The root then initiates the propagation of a token, which will *visit* all the variables of the graph. It starts sending it to the first neighbour and waits for it to come back, before sending it to the next neighbour. When a variable receives a token, it marks the sender as *parent*. When it sends a token, it marks the destination variable as *child*. The token can return either from the variable to whom the token was sent, or from another neighbour, in which case this neighbour is marked as *pseudo-child*. Again, choosing to send the token to the neighbour known to be the most connected is a heuristic likely to create a tree with a lower induced width, hence offering better performance for DPOP.

2.2 DPOP

Two categories of algorithms using DFS trees stand out: ADOPT is an asynchronous search algorithm using DFS trees to perform a top-down search procedure; DPOP uses a different approach: it aggregates solutions from bottom to top of DFS trees and does not search.

DPOP is a distributed version of the bucket elimination algorithm [8]. It has three phases: a DFS tree creation, a `UTIL` propagation and a `VALUE` propagation. At the end of the tree creation, all the variables consistently label each other as parent/child and pseudo-parent/pseudo-child. This serves as a communication structure for the two other phases.

Definition 6 (`UTIL` message). *The `UTIL` message sent by agent i to agent j is a multidimensional matrix, with a dimension per variable in the separator of i. The matrix is filled with the costs of each assignation.*

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 7 (Junction). *Let $f_1 \cdots f_k$ be functions defined over $d_1 \cdots d_k$, the junction $\sum f_i$ is defined over $u = \bigcup d_i$, such that for each variable x in u, $(\sum f_i)(x) = \sum(f_i(x))$.*

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

Definition 8 (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 $\widetilde{f}_x$ of $s - \{x\}$ such that for every assignation $\{y_1 = u_1, \cdots, y_n = u_n\}$ of $s - \{x\}$, $\widetilde{f}_x(u_1, \cdots, u_n) = \min_x f(x, u_1, \cdots, u_n)$.*

The projection is an optimisation operation where the current node picks its optimal value for each possible assignation of the variables in its separator. With the same example, x will minimise the function $(f_1 + \cdots + f_k)$ and create $\widetilde{f}_x$ defined on $sep(x)$, to be sent to its parent. The optimal value of the tree root will thus be the result of the join/project process operated by the root node. From this optimal value, all children can choose their optimum according to the value given to each variable in their separator and to the projection they operated.

This `VALUE` propagation process is initiated by the root of the tree. Each node determines its optimal value based on the values attributed to its separator variables and propagates this value to its children with a `VALUE` message.

3 Symmetries

Symmetries are omnipresent in nature, thus widely used in physics or engineering, making problems easier hence faster to solve. As well, we can profit from symmetries in constraint programming [6, 9] in order to avoid revisiting equivalent assignations. As the induced width of DFS trees directly affects DPOP's performance, deriving implied constraints [5] will not make the resolution faster. Therefore, we assume the problem reformulation approach is the only acceptable way to deal with symmetries: [2] reformulates the problem after a preprocessing step (and breaks symmetries), but we propose here a way to reformulate the problem as we solve it (and exploit symmetries without breaking them).

3.1 Definitions

Definition 9 (Symmetry). *A variable symmetry over a CSP is a mapping that permutes the variables of the problem by pair, while leaving the 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.

The detection of symmetries over centralised CSPs has been studied in [10] with the help of group theory. In a distributed context, all the agents only have a partial view over the problem, and are unable to find symmetries by themselves. Henceforth, the distribution of the definition requires a different approach.

Definition 10 (Partial representation). *A partial representation of a problem p is the restriction of p to a subset of variables $\mathcal{V}$, their neighbour variables, and the constraints involving any variable in $\mathcal{V}$. Each agent naturally owns a partial representation of the problem restricted to its variables.*

Definition 11 (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.*

Proposition 1. *If σ is a partial symmetry over p_1 and p_2, two partial representations of the same CSP, then σ is a partial symmetry over $p_1 \cup p_2$.*

Proof. Let c be a constraint of $p_1 \cup p_2$. If c is a constraint of p_1, then $\sigma(c) \in p_1 \subset p_1 \cup p_2$. As well, if c is a constraint of p_2, then $\sigma(c) \in p_2 \subset p_1 \cup p_2$. Therefore, σ is a symmetry over the union of the problems.

Corollary 1. *If σ is a partial symmetry for each agent involved in a distributed CSP, then it is a symmetry for the global problem.*

Theorem 1. *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.*

Proof. Let σ be a partial symmetry for an agent a. Let $b \neq a$ be an agent of the DCSP. If no variable of b is subject to a permutation in σ, then the problem definition according to b is unchanged through σ. If not, σ is a partial symmetry of b by hypothesis. Thus, according to corollary 1, σ is a symmetry.

Computing in a distributed way the intersection of all the partial symmetries of a given problem as a preprocessing method is a basic method to exploit the symmetries [2]. Hereunder, we propose an algorithm to compute this intersection simultaneously with the `UTIL` propagation process of the DPOP algorithm.

3.2 Symmetries in DFS trees

We present in this paper SymDPOP, an algorithm inspired from DPOP, which is adapted for detecting symmetries in the DFS tree structures and not in the constraint graph. A symmetry in the DFS tree is a symmetry of the constraint graph, but the reverse is not necessarily true.

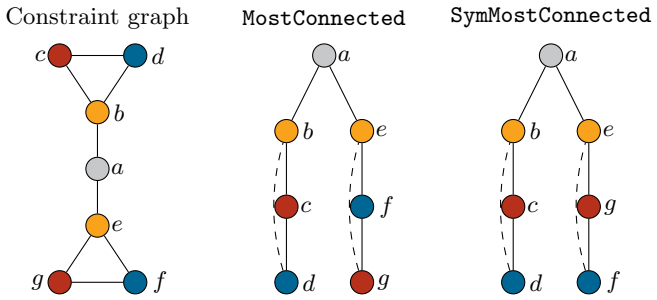

Fig. 2. Symmetric and non-symmetric DFS trees representing a symmetric problem

DPOP appears to be particularly efficient when, during the tree generation, the token is passed in priority to the neighbour which has the largest neighbourhood. In case of equal number of neighbours, the choice among them can be done arbitrarily by, for example, the alphabetical order of the variable names.

However, as symmetry detection focuses on permutations of variables owned by the same agent, we introduce a simple heuristic to choose the next variable in a total order of the *agent owning* it, eg. the alphabetical order of agent names, before the final tie-breaking with variable names. This `SymMostConnected` heuristic is more likely to keep the symmetry of the constraint graph in the DFS tree, as shown on figure 2 where colours represent agents owning the variables.

3.3 SymDPOP

Detecting symmetries in a distributed context is based on the idea of theorem 1. Each agent is able to detect its partial symmetries and if those are also partial symmetries for the neighbours of agents owning variables involved in those symmetries, they are symmetries of the whole problem. While the preprocessing method used in [2] communicates at the agent level, SymDPOP uses the DFS structure produced by DPOP algorithm and involves communication between variables, drawing comparisons between variables owned by the same agent.

Like DPOP, the symmetry detection process starts from the leaves, builds and propagates partial symmetries permuting subtrees, until it reaches the top of the tree. However, each node n only keeps a representation of this partial symmetry σ restricted to $\{n\} \cup sep(n)$, and propagates the representation of σ restricted to $sep(n)$ only.

Definition 12 (`SymUTIL` message). *A `SymUTIL` message is a `UTIL` message to which we attach a set $\mathcal{S}$ of partial symmetries defined as images of each variable in the separator of the `UTIL` message. For each symmetry σ_i of the set $\mathcal{S}$, and for each variable x of the separator, if x is owned by agent a, then $\sigma_i(x)$ is also owned by the same agent a.*

Proposition 2. *It is possible to rebuild all the symmetric `UTIL` messages from a `SymUTIL` message.*

Algorithm 1 Once agents know about their DFS tree

```
for all agent owning leaf variables do
  consider partial symmetries involving those leaves;
  group leaves subject to partial symmetries;
  for all source leaf do
    build SymUTIL and start propagation;
  end for
  for all leaf not involved in any symmetry do
    start UTIL propagation;
  end for
end for
```

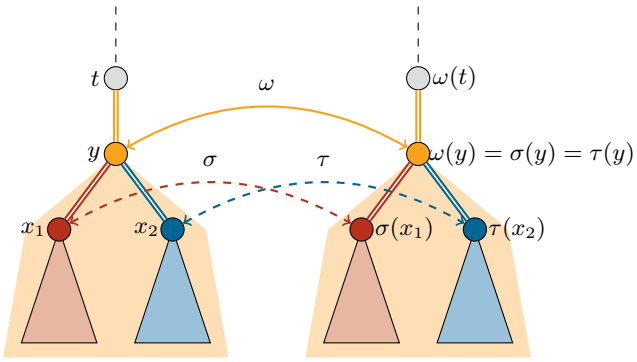

Fig. 3. Consistency

After generating the DFS tree, each agent having leaf variables examines them with their constraints to find those subject to a partial symmetry (algorithm 1). Among those symmetric leaves, it labels one as *source leaf*, and builds a set of partial symmetries $\sigma_1 \cdots \sigma_n$ associating each source leaf to its symmetric leaves, and the separator of the source leaf to its symmetric separators. The source leaves then build a `SymUTIL` message and sends it to their parent.

Definition 13 (Consistency). *Let σ and τ be two partial symmetries of two subproblems p and q. We say that σ and τ are consistent if for each variable x in $\mathcal{D}(p) \cap \mathcal{D}(q), \sigma(x) = \tau(x)$.*

Proposition 3. *Let σ and τ be two partial symmetries on two subproblems p and q. If σ is consistent with τ, there exists a partial symmetry on $p \cup q$.*

This proposition will determine whether and how to propagate the symmetry detection. Intuitively, referring to Figure 3, both nodes x_1 and x_2 are roots of a subtree, for which there exists a partial symmetry σ (resp. τ) associating the subtree to another subtree. If they match on their intersection, which is reduced to $\{y\}$ on the figure, and if we can also swap the other neighbours (here the parents) of y with $\sigma(y) = \tau(y)$, then we can build a symmetry ω associating the

Algorithm 2 On reception of all UTIL/SymUTIL messages

```
if all the received messages are UTIL then
  continue UTIL propagation;
else
  if the problem is still partially symmetric then
    continue SymUTIL propagation;
  else
    regenerate();
  end if
end if
```

Algorithm 3 regenerate()

```
for all SymUTIL received do
  rebuild the UTIL messages from the SymUTIL;
  for all rebuilt UTIL messages do
    continue UTIL propagation;
  end for
end for
```

subtree rooted in y to the subtree rooted in $\omega(y) = \sigma(y) = \tau(y)$. We formalise this idea and extend it to the case of multiple symmetries in the following corollary.

Corollary 2 (SymUTIL propagation). *Let $(x_1, \cdots x_n)$ be sibling variables of the same DFS tree, and $\mathcal{S}_i = (\sigma_{i_1}, \cdots \sigma_{i_k})$ be the set of partial symmetries in the SymUTIL message sent by x_i to y, parent variable of all the (x_i). Let υ be a partial symmetry on the constraints involving y (and $\upsilon(y)$). If by picking one σ_i in each $\mathcal{S}_i$, all those selected σ_i are consistent together and with υ, then there exists a partial symmetry ω on the subproblem represented by y-rooted subtree.*

Proof. Since υ and all σ_i are consistent, there exist a partial symmetry ω on the union of their associated subproblems. ω involves variables which are either descendants of y or in the separator of y. For each variable z in $\mathcal{D}(\omega)$, if z is a variable or linked to a variable from one of the x_i-rooted subtrees, then $\omega(z) = \sigma_i(z)$. Otherwise, it is a neighbour of y and thus in $\mathcal{D}(\upsilon)$, in which case $\omega(z) = \upsilon(z)$.

At the reception of all the SymUTIL messages (algorithm 2), destination variable y cross-checks the consistency of the partial symmetries of its children, and considers a partial symmetry on its separator, building *de facto* a potential υ and ω. If this ω is a partial symmetry, the agent computes only one junction/projection operation for y and sends the appropriate SymUTIL message to its parent. If the partial symmetries are not consistent, the symmetry detection process stops (algorithm 3), regenerates as many UTIL messages as necessary, attributes them to each variable and lets them continue the regular UTIL propagation of DPOP algorithm.

When the UTIL/SymUTIL propagation is finished, each variable, starting from the top of the tree, gets attributed a value. If the top of the tree is in a symmetric

Algorithm 4 On reception of a `SymVALUE` message

assign current variable x and all the $\sigma_i(x)$.
continue `SymVALUE` propagation.

Algorithm 5 On reception of a `VALUE` message

if symmetric separators get the same assignation **then**
  start a unique `SymVALUE` propagation;
else
  continue `VALUE` propagation;
end if

state, it propagates the value down with `SymVALUE` messages, otherwise it uses `VALUE` messages. The content of `VALUE` and `SymVALUE` messages is the same; only the name differs.

Proposition 4. *If variable x sends a `SymUTIL` message, including n symmetries σ_i, and if the values attributed to $sep(x)$ are equal to the ones attributed to $sep(\sigma_i(x))$, then the values attributed to each variable of x-rooted subtree are equal to the values attributed to the image of this subtree through σ_i.*

Indeed, whether the whole problem is symmetric or not, if x sends a `SymUTIL` message, it is the head of a subtree which is partially symmetric to the subtree rooted in $\sigma_i(x)$. If y is a descendent of x, it will choose its value according to the value of its separator. The variables in y's separator are either descendants of x or part of x's separator. Consequently, when a variable x receives a `SymVALUE` message, it assigns the same value to the images of x through all the partial symmetries σ_i (algorithm 4).

When it receives a `VALUE` message (algorithm 5), it keeps propagating unless the variable previously sent a `SymUTIL` message. In that case, the agent starts only one `SymVALUE` propagation for each group of partially symmetric variables which has got the same separator assignation.

3.4 Example on a symmetric problem

We consider the problem between the following agents: c (●) owns c_0, x (●) owns $\{x_1, x_2\}$, y (●) owns $\{y_1, y_2\}$, z (●) owns $\{z_1, z_2\}$. All the variables are defined on $\{0, 1, 2\}$ and are subject to the following constraints: $c_0 \neq x_1, c_0 \neq x_2$, $x_1 \neq y_1, x_2 \neq y_2$, $z_1 \neq y_1, z_2 \neq y_2$, $x_1 \neq z_1, x_2 \neq z_2$.

With the preprocessing method, agent x will detect a partial symmetry $\sigma = (x_1 \rightleftharpoons x_2, y_1 \rightleftharpoons y_2, z_1 \rightleftharpoons z_2)$, and suggest it in order to agents y, z and c, which will agree on the symmetry, and on a reformulation leading to the DFS tree on the left side of Figure 4.

In contrast, SymDPOP starts at the end of the generation of the DFS tree. Agent z finds that σ is a partial symmetry, and sends its constraint in a `SymUTIL` message (double line in the figure), together with σ, to its parent, variable y_1.

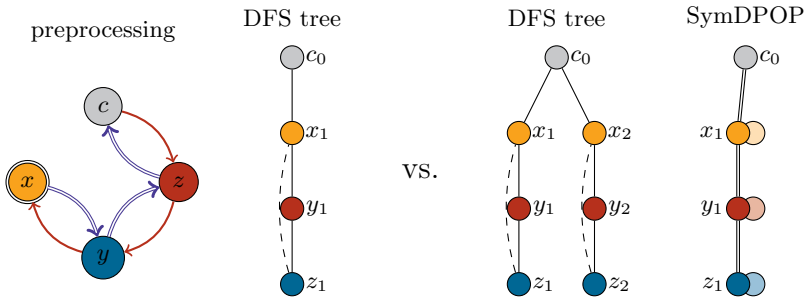

Fig. 4. Difference between preprocessing approach and SymDPOP approach

When agent y gets the message, it considers the permutation of the parents of y_1 and y_2 (x_1 and x_2) which is a partial symmetry consistent with σ. Then, it computes only one junction and one projection operation on y_1, then sends the `SymUTIL` message to x_1. x_1 considers the permutation of the parents of x_1 and of x_2, which are here the same variable c_0: the problem is symmetric.

Agent c receives the `SymUTIL` message and, having no parents, sends a `SymVALUE` message back. When agents receive a `SymVALUE` message, they attribute the value to their variable, and to the image of the variable through σ.

3.5 Example on a non-symmetric problem

We consider the following problem: variables a, b, c and d are all owned by a different agent. Agent t (●) owns variables t_1 and t_2, agent x (●) owns variables x_1 and x_2, agent y (●) owns variables y_1 and y_2 and agent z (●) owns variables z_1 and z_2. Those variables are all defined on $\{0, 1, 2\}$ and are subject to the following constraints: $\forall i, x_i \neq y_i, x_i \neq z_i, x_i \neq t_i$, and $t_1 \neq d$, $t_1 \neq b$, $a \neq b$, $a \neq c$ and $a \neq t_2$. We will refer to Figure 5 as we discuss further about this problem.

First of all, this problem is not globally symmetric: we cannot permute any pair of variables and leave the problem unchanged. However, if agents x, y or z consider their definition of the problem, they can find the following partial symmetries: $\sigma_x = (x_1 \rightleftharpoons x_2, y_1 \rightleftharpoons x_2, z_1 \rightleftharpoons z_2, t_1 \rightleftharpoons t_2)$ for agent x, $\sigma_y = (x_1 \rightleftharpoons x_2, y_1 \rightleftharpoons y_2)$ for agent y, and $\sigma_z = (x_1 \rightleftharpoons x_2, z_1 \rightleftharpoons z_2)$ for agent z. Using DPOP, the `UTIL` message that y_1 would send to x_1 and the one that y_2 would send to x_2 would be identical, and would both be sent between the same agents x and y. Even though the problem is not symmetric, SymDPOP will be able to gather those messages into one `SymUTIL` message.

At the end of the generation of the DFS tree, variables c and d are not subject to partial symmetries, so they follow the regular DPOP procedure and send a `UTIL` message to their parent variable. Then, agent y (resp. z) detects its partial symmetry σ_y (resp. σ_z). As a consequence, variable y_1 (resp. z_1) sends a `SymUTIL` (double line in the figure) message to variable x_1, containing the constraint between x_1 and y_1 (resp. z_1) and the partial symmetry σ_y (resp. σ_z).

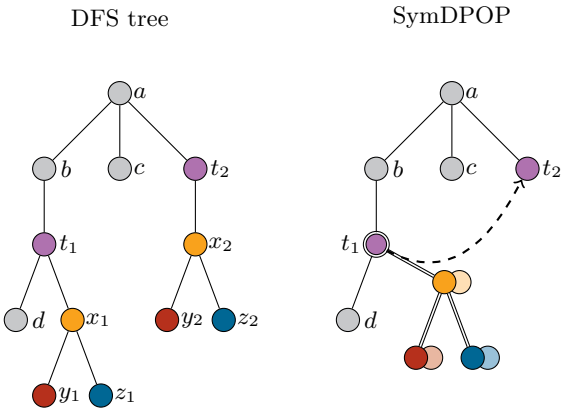

Fig. 5. Partially symmetric problem

When agent x receives the messages, it considers swapping the parent of x_1 (t_1) and of x_2 (t_2) and builds a symmetry consistent with the one received from y and from z. Therefore, it joins the constraints, projects x_1 out and sends the `SymUTIL` message together with σ_x to variable t_1. When it receives the messages, it also receives an `UTIL` message constraining d and $t_1 (\neq t_2)$, hence stopping the `SymUTIL` propagation. Variables t_1 and t_2 are then parents of the root of symmetric subtrees, and the `SymUTIL` message received from agent x is transformed into two `UTIL` messages directed to t_1 and to t_2 (dashed line) who continue the regular `UTIL` propagation process from now on.

On the way down, when agent t gets the two `VALUE` messages, it compares the value assigned to t_1 and t_2. If $t_1 = t_2$ then t_1 sends a `SymVALUE` to x_1, which will assign a value to x_1 and x_2 then sends another `SymVALUE` to agents y and z. If $t_1 \neq t_2$, then t_1 and t_2 both send a `VALUE` message to agent x which will compare the value attributed to x_1 and x_2 and consider as well the choice between a `VALUE` propagation and a `SymVALUE` propagation.

4 Evaluation

4.1 Examples

In this section, we measured the number and volume of messages sent during the process of solving the problems in the past sections. We implemented SymDPOP on FRODO framework [11] for our measurements. We measured the number and volume of messages exchanged for solving the symmetric problem of Figure 4 in Table 1. The biggest part of the total number of messages is used to generate a DFS tree, and both methods work on the same tree, so SymDPOP and DPOP are equal on that aspect. On the `UTIL`/`VALUE` propagation part, however, SymDPOP reformulates the problem dynamically, and cuts the number of messages by 2.

	DPOP	SymDPOP
DFSgeneration	2281	2281
UTIL/VALUE	12	6
total	2293	**2287**

	DPOP	SymDPOP
DFSgeneration	15607	15607
UTIL/VALUE	4278	2794
total	19885	**18401**

	DPOP	SymDPOP
DFSgeneration	3344	3344
UTIL/VALUE	22	19
total	3366	**3363**

	DPOP	SymDPOP
DFSgeneration	24052	24052
UTIL/VALUE	7000	6746
total	31052	**30846**

Table 1. Number (up) and volume (down, in bytes) of messages sent to solve problem of Fig. 4 (left) and problem of Fig. 5 (right)

Be that as it may, in a distributed context, the communication volume has a substantial impact on the performance. Even though lots of messages are sent for generating this DFS tree, those are rather small compared to the messages including a constraint. SymDPOP also cuts the volume of those constraint and value messages. To put the comparison with the `DFSgeneration` phase, SymDPOP cuts almost 10% of the total communication volume here.

Furthermore, one of the main concerns of symmetry treatment is the overhead of symmetry detection. One of the main advantages of SymDPOP is that it does not generate extra messages even if the problem is not symmetric. It uses the regular `UTIL` messages to detect the symmetry and the information added to make a `SymUTIL` messages (a list of variables) is smaller than a whole `UTIL` message (a multidimensional matrix of costs).

As we solved the non-symmetric problem from Figure 5, the values given to t_1 and t_2 were different, and only `VALUE` messages were sent. Table 1 shows the statistics that FRODO gathered. Even though the problem was not symmetric, we saved 3 `UTIL` messages (and few bytes), instead of wasting messages trying to detect a symmetry from the partial symmetries detected by some agents. In contrast, the preprocessing method would use extra messages for the suspected partial symmetries, σ_x, σ_y and σ_z which are not symmetries of the whole problem, and switch back to the regular DPOP method eventually.

4.2 SensorDCSP benchmark

For this evaluation, we used the problem `SensorDCSP` presented in [12] to compare DPOP, SymDPOP, and the preprocessing method of [2]. This problem consists of sensors tracking mobiles moving over a map. At all time, each mobile has to be tracked by exactly three sensors within range. The distribution of the sensors makes this problem naturally distributed. Each variable is a boolean associated to a sensor/mobile couple, and owned by the corresponding sensor. Each constraint over a mobile is distributed over all the sensors detecting it. If two mobiles are close enough to each other, they are detected by the same sensors, thus subject to the same constraints. This makes `SensorDCSP` a convenient benchmark of symmetric distributed problem, albeit not NP-complete.

	50 mobiles			100 mobiles		
	DPOP	preDPOP	SymDPOP	DPOP	preDPOP	SymDPOP
preprocessing		1042			1898	
`DFSgeneration`	30600	**17874**	30600	63648	**30114**	63648
`UTIL/VALUE`	5396	**3212**	**3212**	11048	**5298**	**5298**
total	35996	**22128**	33812	74696	**37310**	68946

Table 2. Number of messages for solving a 25-sensor `SensorDCSP` instance

	50 mobiles			100 mobiles		
	DPOP	preDPOP	SymDPOP	DPOP	preDPOP	SymDPOP
preprocessing		736			1742	
`DFSgeneration`	1696	**1335**	1696	2824	**1889**	2824
`UTIL/VALUE`	1161	**900**	923	1977	**1285**	1406
total	2857	2971	**2619**	4801	4916	**4230**

Table 3. Volume (in kbytes) of messages sent for solving a 25-sensor `SensorDCSP` instance (average on 100 instances)

We compared the executions of this benchmark with FRODO framework in a shared-memory environment[1] and with mpj-express [13] based communication structures on 6 Fujitsu HX600 nodes[2] of the T2K Open Supercomputer [14] for 25 sensors distributed over a map, and with a variable number of mobiles.

Figure 6 shows the computation time averaged on 100 executions. The shared-memory implementation limits the communication effect, therefore SymDPOP improves the computation time by 30 %, versus 50 % for the preprocessing method. However, with a MPI implementation, the preprocessing method hampers the performance, and only SymDPOP manages to yield faster resolution.

We showed in the examples and in the MPI execution that communication volume with SymDPOP was significantly improved. We compared the communication volume in two instances of the problem, with 25 sensors and 50 (and 100) mobiles on Table 3. DPOP and SymDPOP on one hand both generate a complete DFS tree using a large number of small messages. SymDPOP and the preprocessed DPOP use the same number of messages for the `UTIL` and `VALUE` propagation, `SymUTIL` messages being slightly bigger than the `UTIL` ones. However, the preprocessing messages used to detect the symmetries are extremely heavy. Considering the total communication volume, SymDPOP shows a reduction of more than 10 %, whereas the preprocessed DPOP is an actual regress.

5 Conclusions

In this paper, we presented SymDPOP, a new original DPOP algorithm to exploit strict and partial symmetries dynamically as we explored the DFS tree

[1] Core2Duo based Linux, 2GB RAM, `java-6-sun-1.6`

[2] Each node consists of 4 quad-core Opteron 8356, 32GB RAM, `java-6-sun-1.6`

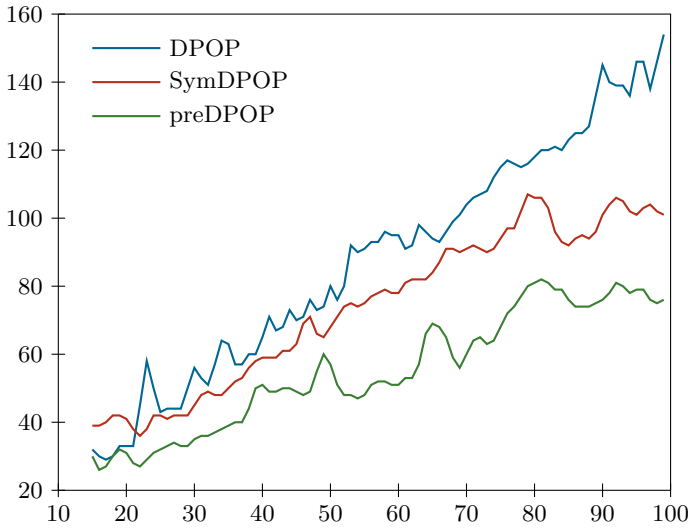

Fig. 6. Computation time (in ms) for solving a {25-x}-SensorDCSP instance (average on 100 instances) on a shared-memory FRODO execution

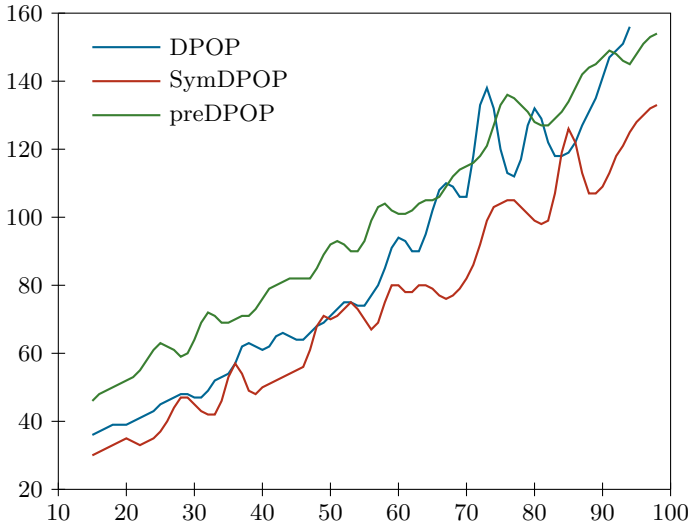

Fig. 7. Computation time (in ms) for solving a {25-x}-SensorDCSP instance (average on 100 instances) on a mpj-express FRODO execution on T2K supercomputer

representation of a problem. The algorithm dynamically reformulates the structure of the problem during `UTIL` propagation phase of the DPOP algorithm. Implemented on FRODO framework with shared memory (resp. MPI) communication structures, SymDPOP cuts down execution time by 30 % (resp. 20 %) and communication volume by 10 %.

One of the most sophisticated parts of this symmetry breaking method is its behaviour on non-symmetric problems. In contrast with other symmetry detection preprocessing methods, SymDPOP never uses extra messages, eliminating any communication overhead which could hamper the resolution of a non-symmetric problem.

References

1. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proc. of 2005 IJCAI. Volume 19. (2005) 266
2. Olive, X., Nakashima, H.: Breaking symmetries in distributed constraint programming problems. In: Proc. of the 9th Int. Workshop on Distributed Constraint Reasoning. (2009)
3. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous agents and Multi-agents systems **3**(2) (2000) 198–212
4. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161**(1-2) (2005) 149–180
5. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. Lecture Notes in Computer Science **2239** (2001) 93–107
6. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: Symmetry definitions for constraint satisfaction problems. Constraints **11**(2) (2006) 115–137
7. Petcu, A., Faltings, B., Parkes, D.: M-DPOP: Faithful distributed implementation of efficient social choice problems. Artificial Intelligence **32** (2008)
8. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1-2) (1999) 41–85
9. Roy, P., Pachet, F.: Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In: Workshop on Non Binary Constraints, ECAI. Volume 98. (1998)
10. Puget, J.F. In: Automatic Detection of Variable and Value Symmetries. Springer Berlin – Heidelberg (2005) 475–489
11. Leaute, T., Ottens, B., Szymanek, R.: FRODO 2.0: An open-source framework for distributed constraint optimization. http://liawww.epfl.ch/frodo/ (2009)
12. Fernandez, C., Bejar, R., Krishnamachari, B., Gomes, C.: Communication and computation in distributed CSP algorithms. In: CP '02: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, London UK, Springer-Verlag (2003) 664–679
13. Baker, M., Carpenter, B., Shafi, A.: MPJ Express: towards thread safe Java HPC. In: Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster 2006). (2006) 1–10
14. Nakashima, H.: T2k open supercomputer: Inter-university and inter-disciplinary collaboration on the new generation supercomputer. Intl. Conf. Informatics Education and Research for Knowledge-Circulating Society (2008) 137–142