A Distributed, Complete Method for Multi-Agent Constraint Optimization

Adrian Petcu[1] and Boi Faltings[1]

Ecole Politechnique Federale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{adrian.petcu, boi.faltings}@epfl.ch
http://liawww.epfl.ch/
Technical Report EPFL/IC/2004/65

Abstract. We present in this paper a new complete method for distributed constraint optimization. This is a utility-propagation method, inspired by the sum-product algorithm [6]. The original algorithm requires fixed message sizes, linear memory, and is time-linear in the size of the problem. However, it is correct only for tree-shaped constraint networks. In this paper, we show how to extend the algorithm to arbitrary topologies using cycle cutsets, while preserving the linear message size and memory requirements. We present some preliminary experimental results on randomly generated problems. The algorithm is formulated for optimization problems, but can be easily applied to satisfaction problems as well.

1 Introduction

Distributed Constraint Satisfaction (DisCSP) was first studied by Yokoo [10] and has recently attracted increasing interest. In distributed constraint satisfaction, variables and constraints are distributed so that each variable and constraint is owned by an agent. Systematic search algorithms for solving DisCSP are generally derived from depth-first search algorithms based on some form of backtracking [9, 11, 12, 7, 3]. Recently, the paradigm of asynchronous distributed search has been extended to constraint optimization by integrating a bound propagation mechanism (ADOPT - [8]).

Backtracking algorithms are very popular in centralized systems because they require very little memory. In a distributed implementation, however, they may not be the best basis since in backtrack search, control shifts rapidly between different variables. Thus, every state change in a distributed backtrack algorithm requires at least one message. Furthermore, in the worst case even in a parallel algorithm there will be exponentially many state changes [5], thus resulting in exponentially many messages.

This leads us to believe that other search paradigms, in particular those based on dynamic programming, may be more appropriate for DisCSP. For example, an algorithm that incrementally computes the set of all partial solutions for all previous variables according to a certain order would only use a linear number of messages. However, the messages could grow exponentially in size, and the algorithm would not have any parallelism.

Recently, the sum-product algorithm [6] has become popular for certain constraint satisfaction problems, for example decoding. It is an acceptable compromise as it combines a dynamic-programming style exploration of a search space with a fixed message

size, and can easily be implemented in a distributed fashion. However, it is correct only for tree-shaped constraint networks. In this paper, we show how to extend the algorithm to arbitrary topologies using cycle cutsets, and report on initial experiments with randomly generated problems. The algorithm is formulated for optimization problems, but can be easily applied to the satisfaction problem by having relations with utility either 0 or 1.

2 Definitions & notation

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

- $\mathcal{A} = \{A_1, ..., A_n\}$ *is the set of agents interested in the problem/solution;*
- $\mathcal{X} = \{X_1, ..., X_m\}$ *is the set of variables/solving agents;*
- $\mathcal{D} = \{d_1, ..., d_m\}$ *is a set of domains of the variables, each given as a finite set of possible values.*
- $\mathcal{R} = \{r_1, ..., r_p\}$ *is a set of relations, where a relation* r_i *is a function* $d_{i1} \times .. \times d_{ik} \rightarrow \Re^+$ *which is expressed by an agent* A_i*, and denotes how much utility that agent assigns to each possible combination of values of the involved variables.*

In this paper we deal with unary and binary relations, being well-known that higher arity relations can also be expressed in these terms with little modifications. In a MCOP, any value combination is allowed; the goal is to find an assignment $\mathcal{X}^*$ for the variables X_i that maximizes the sum of utilities of all the agents $\mathcal{A}$.

A tree-structured problem is a tree network in which we can have several links (constraints) belonging to different agents between two adjacent nodes. Furthermore, unary constraints on each variable are also allowed.

For a node X_k, we define:

- $up - links(X_k)$: constraints between X_k and its parent
- $down - links(X_k)$: constraints between X_k and its children
- $R^1(X_k)$: unary constraints on X_k
- $Ngh(X_k)$: the neighbors of X_k
- R_k: the set of constraints belonging to agent X_k
- $R_k(X_j)$: constraints between X_k and its neighbor X_j

3 Distributed constraint optimization for tree-structured networks

For tree-structured networks (see an example in Figure 1), it is possible to devise polynomial-time complete optimization methods (see the sum-product algorithm for instance [6])

In this problem setting there is a set $\mathcal{X}$ of agents (each agent X_i is responsible for a variable), and a set $\mathcal{A}$ of agents that are interested in the assignments that are made for the variables $\mathcal{X}$. All the agents A_i declare their relations R_i to the agents X_i concerned in those relations (each relation is declared only to the 2 agents X_j and X_k involved -

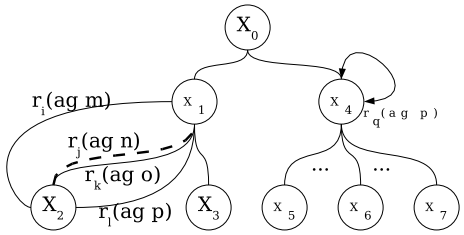

Fig. 1. *Problem example where the underlying constraint graph is a tree.*

assuming binary constraints, or to a single agent in the case of unary constraints). We assume that the resulting constraint graph is a tree.

The "normal" agents - A_i participate in this process only by specifying their relations; in the optimization itself, they have a passive role; only the "variable-agents" will play an active role. Therefore, in the following, while explaining the optimization process, by "agent", or "node" we will mean one of the agents X_i.

In this protocol, agents send messages to each other; the leaf nodes initiate the process, and then the other nodes relay the messages according to the following rule:

Definition 2. *The **k-1 rule**: if node X_i has k neighbors, X_i will send out a message to its k^{th} neighbor only after having received the other k-1 messages, and will send out the rest of k-1 messages after having received the message from the k^{th} neighbor.*

Each agent X_i executes Algorithm 1:

- In the beginning, examine its own relations. All the other agents that are connected through relations with the current node will be its neighbors. During the algorithm an agent communicates only with its neighbors.
- Each agent determines whether it is a leaf in the constraint tree or not (if it has a single neighbor, even if they share multiple constraints) If X_i is a leaf node, then send the *UTIL* message to its only neighbor.
- Wait for incoming messages and respond to them.
  The messages passed in this system are in fact utility vectors; a neighbor X_j of node X_i would send X_i a vector of all the optimal utilities that can be achieved for the subtree rooted at X_i that contains X_j, for each of X_i's possible values (thus, the size of each message is $|dom(X_i)|$
  The agents send messages to their neighbors following the *k-1 rule*. Upon receiving $k-1$ messages from the neighbors, since all of the respective subtrees are disjoint, by summing them up, X_i computes how much utility each of its values gives for the whole set of $k-1$ subtrees. This, together with the relation(s) between X_i and the last neighbor, enable X_i to compute exactly how much utility can be achieved by the entire subtree rooted at the last neighbor and containing X_i, for each of this neighbor's values. Thus, X_i can send to its last (k^{th}) neighbor its *UTIL* message.
  Eventually, the last neighbor would also send its message back to X_i, and at this point X_i would be able to pick the optimal value for itself (as the value that maximizes the sum of the utilities of all subtrees rooted at itself, and of any unary constraints on itself, if any).

At this point, the algorithm is finished for X_i.

Proposition 1. *Algorithm 1 is sound and complete.*

PROOF.

Correctness: since there are no cycles in the problem, it means that all messages that a node X_i receives from its neighbors come from disjoint parts of the constraint problem. They represent exact evaluations of the utility that can be obtained by the subtrees rooted at the sender nodes, for each possible value that X_i can take (can be inferred by induction from the leaves inside the tree) By summing all messages up, X_i has accurate upper bounds on the amount of utility obtained from the whole problem, for each of its values; it is therefore easy to pick the one that gives the maximum utility.

Liveness: again, since there are no cycles in the problem, and all the leaves initiate the message propagation, it is guaranteed that each node will eventually receive *k-1* messages (with k=the number of neighbors) and therefore it will be able to send its k^{th} message. Therefore, it will also receive the final message from the last neighbor, leading to the conclusion of the algorithm for this node. □

Proposition 2. *Algorithm 1 is linear in the number of variables - there are exactly* $2 \times (n-1)$ *messages propagated through the system (where n is the number of agents in the system)*

PROOF. In a tree there are exactly $n-1$ edges between the n nodes of the tree (if less than $n-1$, then we have a set of disconnected problems which we can treat separately, if more, the problem is not a tree anymore). Along each edge, there are exactly 2 messages going through (one from each of the nodes connected through the edge) □

Observations In this algorithm, the agents do not assume any knowledge of the problem structure, and do not have parent-child relationships. All they need to know is whether they are leaf nodes or not (a leaf node has only 1 neighbor), and a way to distinguish between neighbors (ids).

The execution of Algorithm 1 proceeds in an asynchronous fashion from the leaves, traversing the tree and going to other leaves. This means that certain subtrees of the problem proceed faster than others, and it's not always the case that a "child" node is the first to send a *UTIL* message to its "parent" (like it would happen in a centralized setting); it can also happen the other way around (consider the example from Figure 1: it could happen that nodes X_2,X_3 and X_1 finish their processing faster, and X_1 delivers the *UTIL* message toX_0; then, contrary to the centralized setting, X_0 would send its message to X_4 *before* X_4 manages to send its message to X_0). In a sense, the "root" of this tree is dynamically determined, as the single node that happens to receive messages from all its neighbors before being able to send out any message.

4 Distributed constraint optimization for general networks

The scenario is similar to the one for tree networks, except that we can now drop the assumption that the constraint network is a tree. We will show in the following how the previous algorithm must be modified to accommodate this change.

Algorithm 1: *DTREE - Distributed optimization procedure for tree-structured networks.*

DTREE: distributed tree-optimization($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$)
We have a set of agents $X_i \in \mathcal{X}$ that each controls its variable, and a set of agents $A_i \in \mathcal{A}$ that are interested in the assignments of the variables X_i
All agents A_i declare their relations R_i to the subset of agents concerned about those constraints. We assume that the resulting constraint graph is a tree.
Each agent X_i executes:

Initialization(X_i, R_i)
$R_i \leftarrow$ the set of relations binding X_i
$Ngh(X_i) \leftarrow$ the neighbors of X_i (based on R_i)
for all $X_k \in Ngh(X_i)$ **do**
  send $Dom(X_i)$ to X_k
  receive and record $Dom(X_k)$
if $|Ngh(X_i)| == 1$ (i.e. X_i is a leaf node) **then**
  let X_k be the single element in $Ngh(X_i)$
  let $utils_{X_i}(X_k) \leftarrow$ Compute_utils(X_k)
  Send_message($X_k, utils_{X_i}(X_k)$)
$msg\_cnt \leftarrow 0$
activate Message_handler()
return

Message_handler(X_k,$utils_{X_k}(X_i)$)
store X_k, $utils_{X_k}(X_i)$
msg_cnt ++
if $msg\_cnt = |Ngh(X_i)| - 1$ **then**
  let X_j be the only neighbor that did not send $utils_{X_j}(X_i)$ yet
  let $utils_{X_i}(X_j) \leftarrow$ Compute_utils(X_j)
  Send_message($X_j, utils_{X_i}(X_j)$)
else
  if $msg\_cnt = |Ngh(X_i)|$ **then**
    for all $X_l \in \{Ngh(X_i) \setminus X_j\}$ **do**
      let $utils_{X_i}(X_l) \leftarrow$ Compute_utils(X_l)
      Send_message($X_l, utils_{X_i}(X_l)$)

$$v_i^* \leftarrow argmax_{v_i} \left(\sum_{X_l \in Ngh(X_i)} utils_{X_l}(X_i = v_i) + \sum_{r_i \in R^1(X_i)} r_i(v_i) \right)$$

    $X_i \leftarrow v_i^*$
    FINISH_ALGORITHM
return

Compute_utils(X_j)
for all $v_j \in Dom(X_j)$ **do**
  for all $v_i \in Dom(X_i)$ **do**

$$Util_{X_j}(v_i, v_j) \leftarrow \sum_{r_i \in R^1(X_i)} r_i(v_i) + \sum_{r_i \in R_i(X_j)} r_i(v_i, v_j) + \sum_{X_l \in \{Ngh(X_i) \setminus X_j\}} utils_{X_l}(X_i = v_i)$$

  $v_i^*(v_j) \leftarrow argmax_{v_i}(Util_{X_j}(v_i, v_j))$
return a vector $utils_{X_i}(X_j)$ of all $\{Util_{X_j}(v_i^*(v_j), v_j) | v_j \in Dom(X_j)\}$

Send_message($X_j, utils_{X_i}(X_j)$)
send the utils vector to agent X_j
return

First of all, let us consider what would happen if we would directly apply the *DTREE* algorithm to a graph. The fact that the constraint network has cycles breaks the *liveness* argument from Proposition 1 and leads to a deadlock in the execution of the algorithm: messages would still circulate through all the *TREE* parts of the problem, hanging from nodes involved in cycles; however, in a cycle there are no leaf nodes to initiate the message propagation, so the nodes involved in it wait for incoming messages indefinitely.

Based on this observation, we can devise a very simple cycle detection mechanism: whenever some nodes reach a (reasonably chosen) timeout while waiting for (some of) their neighbors to send messages, that means that those nodes are involved in a cycle with the neighbors that did not yet send their messages.

4.1 Cycle cutset

It has been pointed out in the literature [2, 4, 1] that breaking a problem with cycles into cycle-free parts can greatly improve the search performance for centralized, crisp CSPs. In the following, we will try to use this idea to find optimal solutions for *optimization* problems, in a *distributed* fashion.

The basic idea of such a technique would be to identify the nodes involved in cycles, select a subset of these nodes that will act as *cycle cuts*, apply an algorithm similar to *DTREE* to the now cycle-free parts of the problem, and in the end, put together the partial results in a coherent fashion. The rest of this section explains how this can be done.

4.2 Definitions

Node labeling In our model, the nodes of the constraint graph are labeled in one of the following ways:

1. *TREE* (nodes that have at most one path from themselves to at most one *CycleCut* node) - initially only leaf nodes are labeled *TREE*.
2. *Cycle* (nodes that are "between" several CC nodes - there is more than one path from themselves to other CC nodes) - initially all but the leaf nodes are *Cycle*. As a *Cycle* node receives $k-1$ (where k is the number of its neighbors) context-free messages, it turns into a *TREE* node, and sends to the k^{th} neighbor a context-free message.
3. *CycleCut* - *CC* (nodes that are cycle cuts) - initially no node is *CC*; after timeout and negotiation, some become *CC*

Definition 3.

– *disconnected subtree*: a maximal set of interconnected *Cycle* nodes, that connect to the rest of the problem only through *CC* or *TREE* nodes (e.g. $X_i - X_{11} - X_{13} - X_j - X_k$ in Figure 2)

- *cyclic subgraph*: a maximal set of *CC* nodes connected pairwise through at least 2 different *CC* nodes, or through a *disconnected subtree*, together with the *Cycle* nodes from the disconnected subtrees connecting them (for example, X_i and all the lower-right box in Figure 2; a counter-example are Subgraph3 and Subgraph2 in 3, which are disjoint, since they are connected only through X_i)
- *context* of a UTIL message: additional information attached to a UTIL message, specifying under which "assumptions" the respective UTIL message is valid (for instance, a context could be ($X_i = v_2/4, X_k = v_4/7$), meaning that the respective UTIL message is valid when X_i takes its second value out of 4 possible values, and X_k takes its 4^{th} value out out 7 possible values). The context can be null (empty), in which case it means that this message is always valid, without any assumptions. Such messages come from the tree parts of the problem. Messages that circulate inside cyclic subgraphs will have non-empty contexts.
- *context union*: the union of one or more contexts is the union of the sets of variables from all the contexts, with their respective assignments. If one or more variable appears in several contexts, then *it has to have the same assignment in all of them*.

4.3 Topological considerations

In order for the *CC* nodes to know how to treat the incoming messages, it is important for them to have some knowledge of the problem structure. This is important, since in a utility-message propagation algorithm, it is possible that multiple messages coming from the same cycle on different paths are actually duplicates, and should be discarded. On the other hand, messages coming from independent subgraphs should always be considered.

For a categorization of the possible neighborhoods an agent X_i might have, please refer to figure 2. Please note that a "*" denotes the possibility of having 0 or more structures of that kind, a "+" denotes at least one, and a "1" denotes exactly one. The hashed nodes are the nodes that are *CC*, and the others are *TREE* or *Cycle* nodes.

The possible neighborhoods of the node X_i can be categorized as follows:

1. *TREE*: this region is a tree rooted at X_i. X_i's neighbor that is the root of the subtree will eventually send a context-free *UTIL* message.
2. *Subgraph_self*: this region is a part of the graph that contains cycles; however, it suffices to remove X_i to break all these cycles. The probes sent by X_i into this region will return *with the same contexts, which only contain X_i as a CC node*. The contexts contain the same set of ids, but not in the same order (depending on the path they took) Node X_i can differentiate between several independent subgraphs of this type by the set of *Cycle* nodes contained in the context.
3. *Subgraph_safe*: this region may contain one or several other CC nodes and several local cycles; however, apart from the link $X_i - X_j$ there is no other path between X_i and this region.
4. *Subgraph_unsafe*: this region may contain one or more other CC nodes and several local cycles; there are multiple paths from X_i and this region (e.g. $X_i - X_{11}$ and $X_i - X_{12}$). What is important to see is that all these paths will eventually connect. This is the general case, and the previous 2 kinds of cycles are special cases of this one; therefore, in the following, we will discuss only about this kind of cycle.

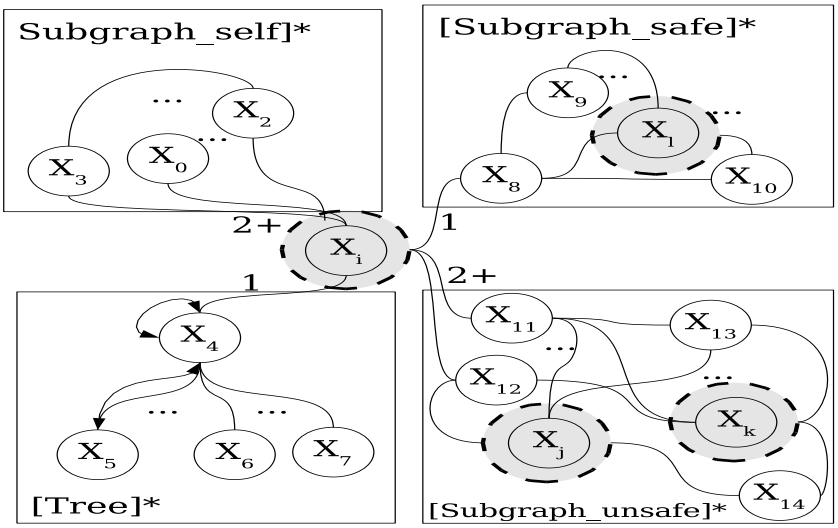

Fig. 2. Categorization of the possible neighborhoods Agent X_i can have, when the underlying constraint problem is a graph.

Topology probing The CC nodes initiate a topology probing process that has as a result the fact that they can categorize their neighboring areas. The probing begins with the *CC* nodes sending out probes to all of their neighbors. Initially the probes have a *context* composed only of the id of the emitting *CC* node. The receiving nodes append their own id to the context of the probes, and then forward them to all their other neighbors. The forwarding stops when reaching *TREE* nodes, or when visiting the same node a second time. For each incoming probe, the *CC* nodes update the largest context that the sending neighbor has sent so far. Upon completion of this procedure (typically after a timeout has been reached), the *CC* nodes sort their neighbors into different sets (cyclic subgraphs) according to their respective largest context; the ones belonging to the same subgraph will necessarily have the same context. They also know their neighborhoods up to the borders of the cyclic subgraphs they are involved in (e.g. in Figure 3, nodes X_i or X_k will know nothing of the Subgraph 1, not even that it exists, since the only contact point between them and any node in Subgraph 1 is the node X_j which will not forward the same probes both ways).

4.4 CyPro - distributed utility probing within a cyclic subgraph

In the most general configuration of a cyclic subgraph, we have a set of *CC* nodes, interconnected through an arbitrary number of disconnected trees (for example, in the lower-right cycle from Figure 2, involving X_i, X_j, and X_k as *CC* nodes, we have 3 disconnected trees: $X_i - X_{12} - X_j - X_k$, $X_i - X_{11} - X_{13} - X_j - X_k$, and $X_j - X_{14} - X_k$). A subgraph like this can be arbitrarily complex. Let us assume for now that there are no links with the outside world (we will relax this condition in section 4.5, and present the complete algorithm)

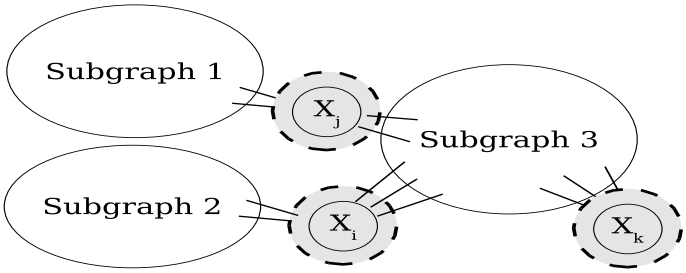

Fig. 3. Problem seen as a meta-tree, composed of cyclic subgraphs connected through *CC* nodes

This algorithm (let us call it *CyPro*) will distributedly generate all the value combinations for all the *CC* nodes involved in this cyclic subgraph, and for each combination, compute the total optimal utility that this assignment yields, provided that the intermediary disconnected trees that lie between the *CC* nodes optimize their values w.r. to this particular assignment of the *CC* nodes. The optimization of the trees is done with a version of *DTREE* extended to support message contexts, therefore the number of messages is linear in the number of arcs of the trees.

During the topology probing phase, each *CC* node received from all its neighbors *TOPO* probes that contained in their context each node in the cyclic subgraph, with the additional *domain size* information for the *CC* nodes involved in this cyclic subgraph. Therefore, each node can easily compute what is the total number of combinations of values required to explore the whole search space: $\prod_{X_i \in CC} |d_i|$. Now, in order to distributedly generate all combinations of values, each node X_i would cycle through all its values for $higher$ times, in each cycle sending out $lower$ probes with the respective value, where

$$higher = max\{1, \prod_{\{X_j \in CC | j > i\}} |d_j|\}, lower = max\{1, \prod_{\{X_j \in CC | j < i\}} |d_j|\}$$

This ensures that all combinations are generated, with the node having the highest id cycling the slowest through its values. *CC* nodes send out their probes to all their neighbors in the subgraph, and wait for replies (they do not forward any messages).

In between the *CC* nodes there are the disconnected trees, composed of *Cycle* nodes that act according to the *k-1 rule*, combining incoming contexts. This ensures that for each value combination that the surrounding *CC* nodes inject in the tree, the results that come out of the tree are optimal with respect to that combination (and contain as context the complete set of *Cycle* nodes from the tree, and the *CC* nodes with their values). Identical results come out from any of the leaves of the tree, so all *CC* nodes connected by that tree have a consistent view of the optimal utility the tree can achieve in that context.

Since the subgraph is arbitrarily complex, it is possible that there is no single node which is connected to all the trees in the subgraph, therefore it is possible that no *CC* node has a global view of the total optimal utility for the current context. In order to overcome this, a ”leader” node is used (it is irrelevant who that leader is, it may be the

node with the highest id in the cycle). Each *CC* node sends the leader a single message that sums up the utilities of the trees that node is involved in, and in which it has the highest id (this ensures that no tree is reported twice). Upon receiving messages from all the *CC* nodes in the subgraph, the leader can sum them up, update its lower bound (thus, it is not needed to store all incoming messages: linear memory requirements), and send back to the other *CC* nodes the result (they can also update their lower bounds, and remember the best local value used in the best context); then a new context is tried, until the last one. At the last context, each *CC* node picks for itself the value that is stored as the best one (from the context that generated the highest utility), and a final round of propagations is initiated, with context-free messages, such that also the *Cycle* nodes within the extended cycle can choose their values.

The algorithm is formally presented in Algorithm 2; informal description:

- If an agent has a single neighbor (even if there are multiple relations to that node), then it labels itself as *TREE*, otherwise as *Cycle*. If X_i is *TREE*, then send the *UTIL* message to its only neighbor.
  The messages passed in this system are the same utility vectors as in *DTREE*, augmented with context information (showing in which context are these vectors valid). If the message is relayed only through *TREE* nodes, then it has an empty context.
- Wait for incoming messages, and respond to them.
- Upon reaching a timeout, X_i realizes it is involved in a cycle, and initiates a negotiation with its neighbors to assume the role of *CycleCut*.
  If the negotiation is successful, X_i becomes *CC*. In the following, the *CC* node will execute two phases: a *topology probing phase*, and a *utility probing phase*.
  Otherwise, negotiation/timeouts repeat until all cycles are broken (detected by the fact that all nodes receive *UTIL* probes/messages).
  If in the end X_i remains a *Cycle* node, then follow the *k-1 rule*.
- *CC* nodes do the *topology probing* (described in section 4.3) and then the *utility probing* (generate all the value combinations of the *CC* nodes involved in the cyclic subgraph and computing the overall optimal utility for each combination)
- *termination*: *TREE* and *Cycle* nodes terminate when the node has received context-free messages from all its neighbors, and *CycleCut* nodes terminate when all the value combinations of their *CC* peers have been explored

Proposition 3. *CyPro is sound and complete.*

PROOF. Follows from the correctness of DTREE (Proposition 1), the fact that all possible value combinations of the cycle cut nodes are tried (a finite number), and that the results of DTREE applied on the disconnected subtrees are combined correctly (only once) by the subgraph leader. □

Overall, for each context, there is a **linear** number of messages generated: $2 \times number\_of\_arcs + 2 \times (k-1)$, where $number\_of\_arcs$ is the number of links (which is less than or equal with the number of relations) in the subgraph, and k is the number of *CC* nodes.

Alternatively, it is possible to cope without any leader, if the *CC* nodes are more "verbose", and send their results to each other ($2 \times number\_of\_arcs + k \times (k-1)$ messages for each context)

Proposition 4. *CyPro has the following complexity:*
$O((dom^k + 1) \times (2 \times number\_of\_arcs + 2 \times (k-1)))$
where dom=domain size, k=size of the cycle cutset and $arcs\_in\_cycle$= the number of arcs in this subgraph.

PROOF. Follows from the discussion above. □

Algorithm 2: *CyPro: distributed utility probing in a cyclic subgraph.*

1: **CyPro(**$Subgraph^k(X_i)$**)**
2: **for all** possible contexts in $Subgraph^k(X_i)$ **do**
3: send out *UTIL* probes with my corresponding value in that context, to all my neighbors
4: wait for incoming *UTIL* probes from all my neighbors in $Subgraph^k(X_i)$
5: duplicates from the same subtree are discarded
6: **if** leader **then**
7: centralize the partial results from all the *CC* peers in $Subgraph^k(X_i)$, and send the total back; update higher bound for my particular value.
8: **else**
9: send the leader the results from the subtrees that I am directly connected to, and in which I am the *CC* node with the highest ID; wait for the total coming from the leader; update higher bound for this particular value of the leader, and remember my own value if bound was improved.
10: At the end, all *CC* nodes know how much utility the whole $Subgraph^k(X_i)$ would get in an optimal assignment for each one of the leader's values, and which one of their values they would pick in that context.

4.5 CyCOpt - distributed cycle-cutset optimization algorithm

We have seen in the previous section that *CyPro* requires fixed message sizes, linear memory, and its message complexity is exponential in the size of the cycle cutset. *CyPro* reduces the complexity from dom^n (equivalent to a standard backtracking) to dom^k (where n=number of nodes in the problem, and k=number of cycle-cut nodes). In the case that the constraint graph is relatively loose, it is likely that $k \ll n$ (a small number of the nodes in the graph are actually cycle-cuts); this would amount to an exponential complexity reduction.

The obvious application of the previous section is to consider the whole problem as an extended cycle, and solve it in the afore mentioned way.

However, in the following, we explore the possibility of further reducing the complexity of the optimization procedure by breaking the problem in *separate* subgraphs, exploring each of them using *CyPro*, and then combining the partial results using a version of *DTREE* that operates at a meta-level, on subgraphs instead of variables. This approach would have the advantage that at a meta-level, the *DTREE* would be linear in the number of subgraphs, and the overall complexity would be the highest complexity of the composing subgraphs.

Some issues need to be considered however, in order to correctly assemble the partial results of *CyPro* applied to the subgraphs:

- *topology*: subgraphs must be independent, connected through at most one *CC* node. That node would play the role of a relay between subgraphs;
- *synchronization*: it is imperative that the *CyPro* be started in a subgraph only after all but one of the externalities (links with other subgraphs through *CC* nodes) have been solved (this is the equivalent *k-1 rule* for the *meta-TREE*);

The first point is already a by-product of the topology-probing phase; it is certain that each *CC* node knows for sure if two subgraphs are independent or not (assuming that there were a link between them in addition to the node itself, a *TOPO* probe is sure to have gone through that link and have returned to the *CC* node, which would have then marked the two subgraphs as the same).

The second one is a little more difficult; in fact it is needed that inside a subgraph there exist a mechanism that allows all the *CC* nodes involved to announce to the other *CC* nodes that they have finished their external *CyPro*s, and now they dispose of accurate and final information about the utility that the rest of the *meta-TREE* can achieve for each of their values. Note that this is completely equivalent to the *k-1* rule for the standard *DTREE*; the difference is that in the standard *DTREE* there was a local decision (each node was receiving all the k-1 messages itself), whereas now we need to implement a *distributed* mechanism that mimics the same functionality.

We solved this problem with a token mechanism: upon solving all of its externalities, a node throws a token in the subgraph; when k-1 (where k is the number of *CC* nodes involved in the subgraph) tokens are received, *CyPro* can be launched. Note that *CC* nodes that are involved in a single subgraph (like X_j and X_k in Figure 2) throw their tokens in from the beginning, since they have no externalities (they are the equivalent of leaf nodes in *DTREE*)

A good strategy is to elect as subgraph leader the last *CC* node that has not yet thrown the token in the subgraph; after *CyPro* is finished in the subgraph, it would be this node that would throw its token in one of its other subgraphs, and start *CyPro* in there, and so on. This synchronization mechanism has the effect that *CyPro*s are starting to cascade, exactly like the *DTREE* propagation that we explained in Section 3.

In the example of Figure 3, node X_k would immediately throw its token in Subgraph 3, X_i in Subgraph 2 and X_j in Subgraph 3. X_k would not start anything in Subgraph 3 because there is a single token in there. *After* having finished *CyPro* in Subgraph 2, X_i would throw its token in Subgraph 3, would see that 2=3-1 tokens exist, and would start *CyPro* in Subgraph 3, etc.

When the last externality of a subgraph is solved, the responsible *CC* node already has complete information for the whole problem (similar to the case in *DTREE* when the last (k^{th}) message is received). It can immediately choose its value, and inform its *CC* peers in all its subgraphs, which in turn will choose theirs, and so on.

The nodes labeled as *TREE* or *Cycle* will execute just as in Section 4.4, sending/relaying messages by the *k-1 rule*. The difference is made by the *CC* nodes that are involved in several subgraphs, which operate in the afore mentioned way.

Proposition 5. *Algorithm 3 is sound and complete.*

PROOF. Follows from Proposition 1, Proposition 3, and the fact that each individual subgraph is explored only when all but one of its externalities are solved (therefore observing the *k-1* rule for the meta-tree). □

Algorithm 3: *CyCOpt: distributed cycle-cutset optimization algorithm.*

CyCOpt($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$**)**
Each agent X_i executes:

Initialization(X_i, R_i**)**
same as in *D-TREE*
if $|Ngh(X_i)| == 1$ (i.e. X_i is a *TREE* node) **then**
  mark X_i as *TREE*, and send *UTIL* message to the single neighbor
else
  mark X_i as *Cycle*
activate Message_handler()
activate Timeout_handler()
return

Timeout_handler()
if (! received any message from at least $|Ngh(X_i)| - 1$ neighbors) **then**
  $Cycle(X_i) \leftarrow \{X_j \in Ngh(X_i) | X_j$ did not send any message yet $\}$
  negotiate cycle_cut with $\forall X_j \in Cycle(X_i)$; set is_cycle_cutset accordingly
  if is_cycle_cutset **then**
    do TOPOLOGY PROBING
    do MAIN PHASE
  else
    reactivate Timeout_handler()

Message_handler()
if X_i is *TREE* or *Cycle* **then**
  relay messages according to the *k-1 rule*
  terminate upon receipt of k context-free messages

TOPOLOGY DEEP PROBING
send out *TOPO* probes to neighbors in $Cycle(X_i)$ and wait for their return
probes are forwarded by *CC*/*Cycle* nodes, collecting in their context the set of visited nodes
upon completion, X_i can categorize all its neighbors in the sets $TREE(X_i)$ (containing all the *TREE* neighbors) and $Cycle^k(X_i)$ (containing all the neighbors in the **independent** cycle $Cycle^k(X_i)$)

MAIN PHASE (*CC* nodes)
if $|Cycles(X_i)| == 1$ **then**
  send my token in my only cycle
for all $Cycle^k(X_i)$ **do**
  wait for c-1 tokens in each cycle (c=the number of *CC* nodes in $Cycle^k(X_i)$, then perform CyPro in the cycle
  when $|Cycles(X_i)| - 1$ cycles have been explored, send my token in the last cycle, and then perform CyPro in there as well
at this point, X_i has complete information from all $Cycle^k(X_i)$, and can choose its optimal value
inform the *CC* peers from all $Cycle^k(X_i)$ about the value chosen
perform a last optimization step in each $Cycle^k(X_i)$ with the chosen value *and context-free UTIL probes*, such that all *Cycle* nodes can also choose their values and terminate.
terminate

Proposition 6. *Algorithm 3 has the following complexity:*
$O((dom^k + 1) \times (2 \times number\_of\_arcs + 2 \times (k - 1)))$
where dom=domain size, k=size of the cycle cutset for the largest subgraph, number_of_arcs = the number of arcs in the largest subgraph

PROOF. As explained above, the problem is broken up in disjoint subgraphs, which are connected through *CC* nodes. Between subgraphs, there is no explicit communication (except for the fact that the node that connects them will depose its token at some point in one of them, when all the rest are done). The difficult problems lie within the subgraphs, and the largest subgraph is the one that gives the overall complexity. Within a subgraph, the message complexity is given by the formula for *CyPro*, so the overall complexity is given by the largest complexity of all subgraphs. When the leader has finally finished as well, another round of $arcs\_in\_cycle \times 2$ messages is required, but this is a one-time, linear number of messages. □

5 Experimental evaluation

We have done some preliminary evaluation of the algorithms on randomly generated optimization problems (weighted graph coloring) with increasing number of variables. We recorded the number of exchanged messages and present the resulting curve in Figure 4. As expected, the number of messages increases with the problem size, which in turn influences the size of the cycle cutset. However, the direct correlation is with the cycle cutset, and not with the problem size, leading us to believe that this method is a good candidate for solving large but sparse problems, where the cycle cutset has manageable sizes.

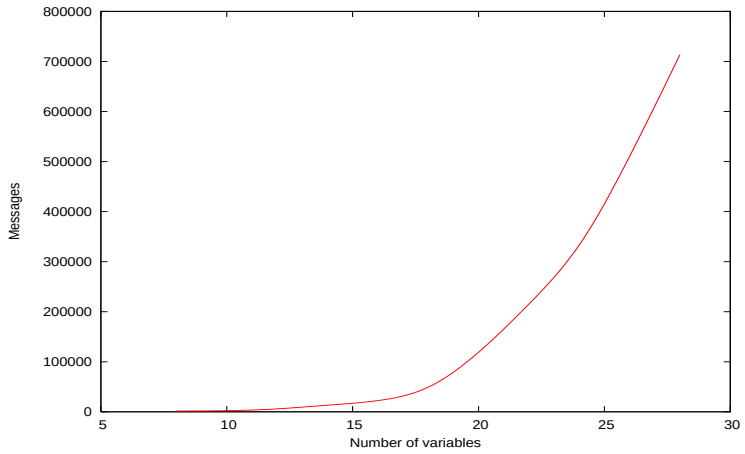

Fig. 4. *Number of messages exchanged while solving problems of increasing size.*

6 Conclusions and future work

We presented in this paper a new complete method for distributed constraint optimization. This method is a utility-propagation method that extends the sum-product algorithm to work on arbitrary topologies using cycle cutsets. It requires fixed message sizes, linear memory, and its message complexity is exponential in the size of the cycle cutset for the largest subgraph in the problem. This method reduces the complexity from dom^n (equivalent to a standard backtracking) to dom^k (*CyPro*) or even $dom^{k'}$ (*CyCOpt*), where n=number of nodes in the problem, k=total number of cycle-cut nodes, and k'=number of cycle-cut nodes in the largest subgraph. For relatively loose problems, it is likely that the inequality $n \gg k \gg k'$ holds, thus our method is likely to produce important complexity reductions.

The algorithm is formulated for optimization problems, but can be easily applied to the satisfaction problem as well.

As future work we consider experimenting with different strategies of selecting the cycle-cut nodes, developing more efficient methods for computation within extended cycles, and more informed topology probing techniques.

References

1. F. Becker and D. Geiger. Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *AI Journal*, 1996.
2. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
3. Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI-98*, pages 219–223, 1998.
4. Arun Jagota and Rina Dechter. Simple distributed algorithms for the cycle cutset problem. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages Pages: 366 – 373, San Jose, California, United States, 1997. ACM, ACM Press New York, NY, USA.
5. Simon Kasif. On the parallel complexity of some constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-86*, pages 349–353, Philadelphia, PA, 1986.
6. Frank R. Kschischang, Brendan Frey, and Hans Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 2001.
7. Amnon Meisels and Roie Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2003, Acapulco, Mexico*, 2003.
8. P. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization, 2003.
9. Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, Austin, Texas, 2000.
10. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.
11. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem - formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
12. Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.