

A Scalable Method for Multiagent Constraint Optimization

Adrian Petcu and Boi Faltings
{adrian.petcu, boi.faltings}@epfl.ch
<http://liawww.epfl.ch/>

Artificial Intelligence Laboratory
Ecole Polytechnique Fédérale de Lausanne (EPFL)
IN (Ecublens), CH-1015 Lausanne, Switzerland
Technical Report IC/2005/002

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 [Kschischang *et al.*, 2001]. The original algorithm requires fixed message sizes, linear memory and linear time in the size of the problem. However, it is correct only for tree-shaped constraint networks. In this paper, we show how to extend that algorithm to arbitrary topologies using a pseudotree arrangement of the problem graph. We compare our algorithm with "standard" backtracking algorithms, and present experimental results. For some problem types we report orders of magnitude less messages, and even the ability to deal with arbitrary large problems. Our 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 [Yokoo *et al.*, 1992] and has recently attracted increasing interest. In distributed constraint satisfaction 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 [Silaghi *et al.*, 2000; Yokoo *et al.*, 1998; Yokoo and Hirayama, 2000; Meisels and Zivan, 2003; Hamadi *et al.*, 1998]. Recently, the paradigm of asynchronous distributed search has been extended to constraint optimization by integrating a bound propagation mechanism (ADOPT - [Modi *et al.*, 2003]).

In general, optimization problems are much harder to solve than DisCSP ones, as the goal is not just to find *any* solution, but the *best* one, thus requiring more exploration of the search space. The common goal of all distributed algorithms is to minimize the number of messages required to find a solution.

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. Every state change in a distributed backtrack algorithm requires at least one message; in the worst

case, even in a parallel algorithm there will be exponentially many state changes [Kasif, 1986], thus resulting in exponentially many messages. So far, this has been a serious drawback for the application of distributed algorithms in the real world, especially for optimization problems (also noted in [Maheswaran *et al.*, 2004]).

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 [Kschischang *et al.*, 2001] has been proposed 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 a pseudotree arrangement of the problem graph, and report on experiments with randomly generated problems. The algorithm is formulated for optimization problems, but can be easily applied to satisfaction problems by having relations with utility either 0 (for allowed tuples) or negative values (for disallowed tuples). Utility maximization produces a solution if there is an assignment with utility 0.

2 Definitions & notation

A discrete *multiagent constraint optimization problem* (MCOP) is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ such that:

- $\mathcal{X} = \{X_1, \dots, X_m\}$ is the set of variables/agents;
- $\mathcal{D} = \{d_1, \dots, d_m\}$ is a set of domains of the variables, each given as a finite set of possible values.
- $\mathcal{R} = \{r_1, \dots, r_p\}$ is a set of relations, where a relation r_i is a function $d_{i1} \times \dots \times d_{ik} \rightarrow \mathbb{R}^+$ which denotes how much utility is assigned 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.

For a node X_k , we define: $R^i(X_k)$ = constraints of arity i on X_k (where i is 1 or 2); $Ngh(X_k)$ = the neighbors of 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, polynomial-time complete optimization methods have been developed (e.g. the sum-product algorithm [Kschischang *et al.*, 2001] and the *DTREE* algorithm from [Petcu and Faltings, 2004]).

In *DTREE*, the agents send *UTIL* messages (utility vectors) to their parents. A child X_l of node X_k would send X_k a vector of the optimal utilities $u_{X_l}^*(v_k^j)$ that can be achieved by the subtree rooted at X_l plus $R_l(X_k)$, and are compatible with each value v_k^j of X_k .

For the leaf nodes it is immediate to compute these valuations by just inspecting the constraints they have with their single neighbors, so they initiate the process. Then each node X_i relays these messages according to the following process:

- Wait for *UTIL* messages from all children. 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 subtree rooted at itself. This, together with the relation(s) between X_i and its parent X_j , enables X_i to compute exactly how much utility can be achieved by the entire subtree rooted at X_i , taking into account compatibility with each of X_j 's values. Thus, X_i can send to X_j its *UTIL* message. X_i also stores its optimal values corresponding to each value of X_j .
- If root node, X_i can compute the optimal overall utility corresponding to each one of its values (based on all the incoming *UTIL* messages), pick the optimal one, and send a *VALUE* message to its children, informing them about its decision.

Upon receipt of the *VALUE* message from its parent, each node is able to pick the optimal value for itself (as the previously stored optimal value corresponding to the value its parent has chosen), and pass it on to its children. At this point, the algorithm is finished for X_i .

The correctness of this algorithm was shown in the original paper, as well as the fact that it requires a linear number of messages.

4 Distributed constraint optimization for general networks

To apply a *DTREE*-like algorithm to a cyclic graph, we first need to arrange the graph as a pseudotree (it is known that this arrangement is possible for any graph).

4.1 Pseudotrees

Definition 1 A pseudo-tree arrangement of a graph G is a rooted tree with the same vertices as G and the property that

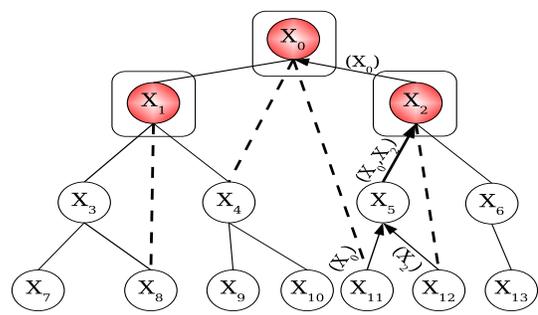


Figure 1: Example of a pseudotree arrangement.

adjacent vertices from the original graph fall in the same branch of the tree (e.g. X_0 and X_{11} in Figure 1).

Pseudotrees have already been investigated as a means to boost search ([Freuder, 1985; Freuder and Quinn, 1985; Dechter, 2003; Schiex, 1999]). The main idea with their use in search, is that due to the relative independence of nodes lying in different branches of the pseudotree, it is possible to perform search in parallel on these independent branches.

We define the following elements (refer to Figure 1):

- $P(X)$ - the *parent* of a node X : the single node higher in the hierarchy of the pseudotree that is connected to the node X directly through a tree edge (e.g. $P(X_4) = X_1$)
- $C(X)$ - the *children* of a node X : the set of nodes lower in the pseudotree that are connected to the node X directly through tree edges (e.g. $C(X_1) = \{X_3, X_4\}$)
- $PP(X)$ - the *pseudo-parents* of a node X : the set of nodes higher in the pseudotree that are connected to the node X directly through back-edges ($PP(X_8) = \{X_1\}$)
- $PC(X)$ - the *pseudo-children* of a node X : the set of nodes lower in the hierarchy of the pseudotree that are connected to the node X directly through back-edges (e.g. $PC(X_1) = \{X_8\}$)

In the example from Figure 1 one can see that some of the edges of the original graph are not part of the spanning tree (otherwise the problem would be a tree). We call such edges *back-edges* (e.g. the dashed edges $8 - 1$, $12 - 2$, $4 - 0$), and the other ones *tree edges*. We call a path in the graph that is entirely made of tree edges, a *tree-path*. A *tree-path associated with a back-edge* is the tree-path connecting the two nodes involved in the back-edge (please note that since our arrangement is a pseudotree, such a tree path is always included in a branch of the tree).

For each back-edge, the node higher in the hierarchy that is involved in that back-edge is called the *back-edge handler* (in Figure 1, the dark nodes 0, 1 and 2 are *handlers*).

As it is already known, a DFS (depth-first search) tree is also a pseudotree (although the inverse does not always hold). So, a DFS tree obtained from the DFS traversal of the graph starting from one of the nodes (chosen through a distributed leader election algorithm) will do just fine. Due to the lack of space we do not present here a procedure for the creation of a DFS tree, and refer the reader to techniques like [Gallager *et al.*, 1979; Barbosa, 1996; Hamadi *et al.*, 1998].

$X_8 \rightarrow X_3$	$X_3 = v_3^0$	$X_3 = v_3^1$...	$X_3 = v_3^{m-1}$
$X_1 = v_1^0$	$u_{X_8}^*(v_1^0)$	$u_{X_8}^*(v_1^0)$...	$u_{X_8}^*(v_1^0)$
...
$X_1 = v_1^{n-1}$	$u_{X_8}^*(v_1^{n-1})$	$u_{X_8}^*(v_1^{n-1})$...	$u_{X_8}^*(v_1^{n-1})$

Table 1: *UTIL* message sent from X_8 to X_3 , in Figure 1

4.2 The *DPOP* algorithm

The algorithm has 3 phases. First, the agents establish the pseudotree structure (see section 4.1) to be used in the following two phases. The next two phases are the *UTIL* and *VALUE* propagations, which are similar to the ones from *DTREE* - section 3.

Please refer to Algorithm 1 for a formal description of the algorithm, and to the rest of this section for a detailed description of the *UTIL* phase. The *VALUE* phase is the same as in *DTREE*.

UTIL propagation

As in *DTREE*, the *UTIL* propagation starts from the leaves of the pseudotree and propagates up the pseudotree, only through the tree edges. It is easy for an agent to identify whether it is a leaf in the pseudotree or not: it must have a single neighbor connected through a tree edge (e.g. X_7 to X_{13} in Figure 1).

In a tree network, a *UTIL* message sent by a node to its parent is dependent only on the subtree rooted at the respective node (no links to other parts of the tree), and the constraint between the node and its parent. For an example see Figure 1, and consider the message ($X_6 \rightarrow X_2$). This message is clearly dependent only on the target variable X_2 , since there are no links between X_6 or X_{13} and any node above X_2 .

In a network with cycles (each back-edge in the pseudotree produces a cycle), a message sent from a node to its parent may also depend on variables above the parent. This happens when there is a back-edge connecting the sending node with such a variable. For example, consider the message ($X_8 \rightarrow X_3$) in Figure 1. We see that the utilities that the subtree rooted at X_8 can achieve are not dependent only on its parent X_3 (as for $X_6 \rightarrow X_2$). As X_8 is connected with X_1 through the backedge $X_8 \rightarrow X_1$, X_8 must take into account this dependency when sending its message to X_3 .

This is where the dynamic programming approach comes into play: X_8 will compute the optimal utilities its subtree can achieve for each value combination of the tuple $\langle X_3, X_1 \rangle$. It will then assemble a message as a hypercube with 2 dimensions (one for the target variable X_3 and one for the back-edge handler X_1), and send it to X_3 (see Table 1).

This is the key difference between *DTREE* and *DPOP*: messages travelling through the network in *DTREE* always have a single dimension (they are linear in the domain size of the target variable), whereas in *DPOP*, messages have multiple dimensions (one for the target variable, and another one for each context variable).

Combining messages - dimensionality increase

Let us consider the example from Figure 1: X_5 receives 2 messages from its children X_{11} and X_{12} ; the message from

Algorithm 1: *DPOP* - Distributed pseudotree-optimization procedure for general networks.

```

1: DPOP( $\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$ )
   Each agent  $X_i$  executes:
2:
3: Phase 1: pseudotree creation
4: elect leader from all  $X_j \in \mathcal{X}$ 
5: elected leader initiates pseudotree creation
6: afterwards  $X_i$  knows  $P(X_i)$ ,  $PP(X_i)$ ,  $C(X_i)$  and  $PC(X_i)$ 
7: Phase 2: UTIL message propagation
8: if  $|Children(X_i)| == 0$  (i.e.  $X_i$  is a leaf node) then
9:    $UTIL_{X_i}(P(X_i)) \leftarrow \text{Compute\_utils}(P(X_i), PP(X_i))$ 
10:  Send_message( $P(X_i)$ ,  $UTIL_{X_i}(P(X_i))$ )
11: activate UTIL_Message_handler()
12: Phase 3: VALUE message propagation
13: activate VALUE_Message_handler()
14: END ALGORITHM
15:
16: UTIL_Message_handler( $X_k, UTIL_{X_k}(X_i)$ )
17: store  $UTIL_{X_k}(X_i)$ 
18: if UTIL messages from all children arrived then
19:   if Parent( $X_i$ ) == null (that means  $X_i$  is the root) then
20:      $v_i^* \leftarrow \text{Choose\_optimal}(null)$ 
21:     Send VALUE( $X_i, v_i^*$ ) to all  $C(X_i)$  and  $PC(X_i)$ 
22:   else
23:      $UTIL_{X_i}(P(X_i)) \leftarrow \text{Compute\_utils}(P(X_i), PP(X_i))$ 
24:     Send_message( $P(X_i)$ ,  $UTIL_{X_i}(P(X_i))$ )
25:   return
26:
27: VALUE_Message_handler( $X_k, v_k$ )
28: add  $X_k = v_k$  to agent_view
29: if VALUE messages came from  $P(X_i)$  and all  $PP(X_i)$  then
30:    $X_i \leftarrow v_i^* = \text{Choose\_optimal}(agent\_view)$ 
31:   Send VALUE( $X_i, v_i^*$ ) to all  $C(X_i)$  and  $PC(X_i)$ 
32:
33: Choose_optimal(agent_view)
34:

$$v_i^* \leftarrow \underset{v_i}{\text{argmax}} \sum_{X_l \in C(X_i)} UTIL_{X_l}(v_i, PP(X_i), P(X_i))$$

35: return  $v_i^*$ 
36:
37: Compute_utils( $P(X_i)$ ,  $PP(X_i)$ )
38: for all combinations of values of  $X_k \in PP(X_i)$  do
39:   let  $X_j$  be Parent( $X_i$ )
40:   similarly to DTREE, compute a vector  $UTIL_{X_i}(X_j)$ 
   of all  $\{Util_{X_j}(v_i^*(v_j), v_j) | v_j \in Dom(X_j)\}$ 
41: assemble a hypercube  $UTIL_{X_j}(X_i)$  out of all these
   vectors (with  $|PP(X_i)| + 1$  dimensions).
42: return  $UTIL_{X_j}(X_i)$ 

```

X_{11} has X_0 as context, and the one from X_{12} has X_2 as context. Both have one dimension for X_5 (target variable) and one dimension for their context variable (X_0 and X_2 respectively), therefore, their dimensionality is 2. X_5 needs to send out its message to its parent (X_2). X_5 considers all possible values of X_2 , and for each one of them, all combinations of values of the context variables (X_0 and X_2) and X_5 are considered; the values of X_5 are always chosen such that the optimal utilities for each tuple $\langle X_0 \times X_2 \times X_2 \rangle$ are achieved. Note that since X_2 is both a context variable and the target variable, the resulting message has 2 dimensions, not 3.

One can think of this process as the cross product of messages $X_{11} \rightarrow X_5$ and $X_{12} \rightarrow X_5$ resulting in a hypercube with dimensions X_0 , X_2 and X_5 , followed by a projection on the X_5 axis, which retains the optimal utilities for the tuples $\langle X_0 \times X_2 \rangle$ (optimizing w.r.t. X_5 given X_0 and X_2).

Collapsing messages - dimensionality decrease

Whenever a multi-dimensional *UTIL* message reaches a target variable that occupies one dimension in the message (a back-edge handler), the target variable optimizes itself out of the context, and the outgoing message loses the respective dimension.

We can take the example of X_1 , which is initially present in the context of the message $X_8 \rightarrow X_3$: once the message arrives at X_1 , since X_1 does not have any more influence on the upper parts of the tree, X_1 can "optimize itself away" by simply choosing the best value for itself, for each value of its parent X_0 (the normal *DTREE* process). Thus, one can see that a back edge handler (X_1 in our case) appears as an extra dimension in the messages travelling from the lower end of the back edge (X_8) to itself, through the tree path associated with the back edge ($X_8 \rightarrow X_3 \rightarrow X_1$).

5 Complexity analysis

The message propagation is similar to *DTREE*, so the number of messages is linear. The complexity of this method lies in the size of the messages (they are exponential in their dimension). We have seen that a back-edge only influences the dimensionality of the messages travelling through its associated tree-path, and otherwise has no influence on other parts of the pseudotree. It follows that increases of dimensionality can happen only when such tree-paths overlap for at least one edge. Furthermore, if we consider the case of several back-edges having the same handler, we see that their tree-paths necessarily overlap, but this produces only an increase of 1 dimension (the handler variable itself), and not one for each back-edge. Thus, it is easy to see that the overall complexity is exponential in the maximal number of overlaps between tree-paths associated with back-edges that have different handlers. As an example, consider Figure 1: the overall complexity is given by the two back-edges $X_{11} \rightarrow X_0$ and $X_{12} \rightarrow X_2$, whose associated tree-paths intersect on the edge $X_5 \rightarrow X_2$.

We will show in the following that the maximal message size can be characterized by the *induced width* of the graph ordered according to the DFS traversal of the pseudotree. Dechter ([Dechter, 2003], chapter 4, pages 86-88) gives us a way to obtain this parameter: "the fillup method". First, we

build the *induced graph* from the original graph as follows: we choose an ordering of the graph and process the nodes recursively (bottom up) along the chosen order; when a node is processed, all its parents are connected (if not already connected). The *induced width* is the maximum number of parents of any node in the induced graph.

If we consider as an ordering the DFS traversal of the pseudotree, we easily see that any given node cannot have more than one parent, except when there is at least one back-edge connecting it with one (or more) pseudoparents.

If no node in the pseudotree has more than one parent, the graph is obviously a tree (no extra edges). Dechter showed that in this case, the width of the graph is 1. This case reduces to *DTREE*, which requires linear time and memory.

If there is a node X_i with more than one ancestor, the fillup method connects X_i 's parent (X_k) with its pseudoparent X_j (the handler of the back-edge $X_i \rightarrow X_j$). Then, X_k is processed, which now has two ancestors: its own parent, and X_j (X_k was connected with X_j in the previous step). Therefore, another link is added between X_k and X_j . Recursively, the process repeats along the tree-path between X_i and X_j , adding one edge between X_j and each node along that path.

We see that the width of the nodes along that tree-path has increased by 1 (giving an increase of 1 also for the induced width of the graph), and that nothing else is affected (edges are added *only* between X_j and the nodes along the tree-path $X_i \rightarrow X_j$).

Let us consider what would happen if there were another back-edge in the pseudotree. There are 3 possible cases:

1. the associated tree-paths of the back-edges do not overlap: the fill-up method adds one edge to all nodes along the two tree-paths, from their lower-ends all the way up the pseudotree to their respective back-edge handlers; however, since the tree-paths are disjoint, each node increases its width only by one; therefore the induced width of the graph also increases only by one;
2. the associated tree-paths of the back-edges overlap, and they have the same back-edge handler: edges are added only once (one handler) from the handler to the nodes on the tree-paths of the back-edges, even when they overlap; therefore, there is an increase in width only by one;
3. the associated tree-paths of the back-edges overlap, and they have different back-edge handlers.

Consider an example tree-path $X_1 - X_2 - X_3 - X_4 - X_5 - X_6$; suppose there are 2 back-edges $X_5 \rightarrow X_2$, and $X_6 \rightarrow X_3$. We see that their respective tree-paths overlap on the edges $X_5 \rightarrow X_4$ and $X_4 \rightarrow X_3$. The recursive process begins from the lowest node in the ordering (X_6), and starts adding edges: $X_3 \rightarrow X_5$ and $X_2 \rightarrow X_4$. The result is that we have two back-edges ($X_5 \rightarrow X_2$ and $X_6 \rightarrow X_3$) with overlapping tree-paths and different back-edge handlers (X_2 and X_3); thus, the induced width of the graph is given by the width of node 5, which is 3.

We can conclude that the width of the graph induced by the ordering given by the DFS traversal of the pseudotree is actually given by the number of back-edges with different handlers whose corresponding tree-paths overlap.

Theorem 1 *Algorithm 1 requires a linear number of messages, the largest one being space-exponential in the induced width of the pseudotree.*

PROOF.

There are $n - 1$ *UTIL* messages (one through each tree-edge), and then m *VALUE* messages (one through each one of the m edges of the graph).

As for the second part of the claim (maximal message size equals induced width), we saw in the previous section that both these quantities are equal to the maximal number of overlaps between tree-paths associated with back-edges that have different handlers. Thus, we can conclude that the largest message is exponential in the width of the graph induced by the pseudotree ordering. \square

Exponential size messages are not necessarily a problem in all setups (depending on the resources available and on the induced width - low width problems generate small messages!)

However, when the maximum message size is limited, one can serialize big messages using a simple technique: the back-edge handlers ask explicitly for valuations for each one of their values *sequentially*, so each message can have customizable size.

A workaround against exponential memory is possible by renouncing exactness, and propagating valuations for the best/worst value combinations (upper/lower bounds) instead of all combinations.

6 Comparison with other approaches

Schiex [Schiex, 1999] notes the fact that so far, pseudotree arrangements have been mainly used for *search* procedures (essentially backtrack-based search, or branch-and-bound for optimization). All these procedures have a worst case complexity exponential in the depth of the pseudotree arrangement (basically because all the variables on the longest branch from root to a leaf have to be instantiated sequentially, and all their value combinations tried out).

Our approach exhibits a worst case complexity exponential in the width of the graph induced by the pseudotree ordering.

Arnborg shows in [Arnborg, 1985] that finding a min-width ordering of a graph is NP-hard; however, the DFS traversal of the graph has the advantage that it produces a good approximation, and is really easy to implement in a distributed context.

It was shown in [Bayardo and Miranker, 1995] that there are ways to obtain shallow pseudotrees (within a logarithmic factor of the induced width), but these require intricate heuristics like the ones from [Freuder and Quinn, 1985; Maheswaran *et al.*, 2004], which have so far not been adapted to a distributed setting (also noted by the authors of the second paper).

Furthermore, it was shown by Dechter in [Dechter and Fattah, 2001] that the induced width is always less than or at most equal with the pseudotree height; thus we can conclude theoretically that our algorithm will always do at least as well as a pseudotree backtrack-based algorithm on the same pseudotree ordering.

It is also to our advantage that our algorithm will nicely do with a simple DFS ordering, without the need to employ sophisticated heuristics to minimize its depth, because the depth of the pseudotree is irrelevant to the complexity. To see this fundamental difference, consider a problem that is a ring with 100 nodes. A DFS ordering of such a graph would yield a pseudotree with height 100, and one back edge, thus induced width 2. A backtracking algorithm would be exponential in 100, whereas our algorithm is exponential in 2.

7 Experimental evaluation

One of our experimental setups is the sensor grid testbed from [Bejar *et al.*, 2005]. Briefly, there is a set of targets in a sensor field, and the problem is to allocate 3 different sensors to each target. This is a NP-complete resource allocation problem.

In [Bejar *et al.*, 2005], random instances are solved by AWC (a complete algorithm for constraint *satisfaction*). The problems are relatively small (100 sensors and maximum 18 targets, beyond which the problems become intractable). Our initial experiments with this setup solve to optimality problems in the 400 sensors grid, with up to 40 targets.

Another setup is the one from [Maheswaran *et al.*, 2004], where there are corridors composed of squares which indicate areas to be observed. Sensors are located at each vertex of a square; in order for a square to be "observed", all 4 sensors in its vertices need to be focused on the respective square. Depending on the topology of the grid, some sensors are shared between several squares, and they can observe only one of them at a time. The authors test 4 improved versions of ADOPT (current state of the art) on 4 different scenarios, where the corridors have the shapes of capital letters L, Z, T and H (their results and a comparison with *DPOP* are in Table 2). One can see the dramatic reduction of the number of messages required (in some cases orders of magnitude), even for these very small problem instances (16 variables). The explanation is that our algorithm always produces a linear number of messages. This fact translates into our algorithm's ability to solve arbitrarily large instances of this particular kind of real-world problems.

There is of course a question about the size of the messages. However, these problems have graphs with very low induced widths (2), basically given by the intersections between corridors. Thus, our algorithm employs linear messages in most of the parts of the problems, and only in the intersections are created 2 messages with 2 dimensions (in this case with 64 values each). In real world scenarios, sending a few larger messages is preferable to sending a lot of small messages because of the much lower overheads implied (differences can go up to orders of magnitude speedups).

8 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 tree propagation algorithms like the sum-product algorithm or *DTREE* to work on arbitrary topologies using a pseudotree structure. It requires a linear number of messages, the largest one being exponential in the induced width along the particular pseudotree cho-

Algo/Scenario	Test L	Test Z	Test T	Test H
MCN , No Pass	626.4	1111.64	1841.28	1898.04
MLSP, No Pass	597.88	663.32	477.56	679.36
MCN , Pass	95.67	101.90	94.93	258.07
MLSP , Pass	81.77	91.5	107.77	255.2
DPOP	30	30	18	30

Table 2: DPOP vs 4 ADOPT versions: number of messages.

sen. This method reduces the complexity from dom^n (standard backtracking) to dom^w , where n =number of nodes in the problem and w =the induced width along the particular pseudotree chosen. For loose problems, $n \gg w$ holds and our method produces important speedups (even orders of magnitude fewer messages). Our experiments show that our method is the first one to be able to handle effectively arbitrarily large instances of practical problems while using a linear number of messages.

Finding the minimum width pseudotree is a NP-complete problem, so in our future work we will investigate heuristics for finding low width pseudotrees.

9 Acknowledgements

We would like to thank Rina Dechter and Radu Marinescu for insightful discussions and Jonathan Pearce for providing us with experimental data from sensor networks simulations.

References

- [Arnborg, 1985] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, (25):2–23, 1985.
- [Barbosa, 1996] Valmir Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.
- [Bayardo and Miranker, 1995] Roberto Bayardo and Daniel Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*, 1995.
- [Bejar *et al.*, 2005] Ramon Bejar, Cesar Fernandez, Magda Valls, Carmel Domshlak, Carla Gomes, Bart Selman, and Bhaskar Krishnamachari. Sensor networks and distributed CSP: Communication, computation and complexity. *Artificial Intelligence*, 161(1-2):117–147, 2005.
- [Dechter and Fattah, 2001] Rina Dechter and Yousri El Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2):93–118, 2001.
- [Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
- [Freuder and Quinn, 1985] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI-85*, 1985.
- [Freuder, 1985] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 1985.
- [Gallager *et al.*, 1979] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum weight spanning trees. Technical Report LIDS-P-906-A, Massachusetts Inst. of Technology, 1979.
- [Hamadi *et al.*, 1998] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI-98*, pages 219–223, 1998.
- [Kasif, 1986] 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.
- [Kschischang *et al.*, 2001] Frank R. Kschischang, Brendan Frey, and Hans Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions On Information Theory*, 2001.
- [Maheswaran *et al.*, 2004] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the realworld: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS*, 2004.
- [Meisels and Zivan, 2003] Amnon Meisels and Roie Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2003, Acapulco, Mexico*, 2003.
- [Modi *et al.*, 2003] P. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization, 2003.
- [Petcu and Faltings, 2004] Adrian Petcu and Boi Faltings. A distributed, complete method for multi-agent constraint optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004) in Toronto, Canada*, September 2004.
- [Schiex, 1999] Thomas Schiex. A note on CSP graph parameters. Technical report, INRA, 1999.
- [Silaghi *et al.*, 2000] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, Austin, Texas, 2000.
- [Yokoo and Hirayama, 2000] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.
- [Yokoo *et al.*, 1992] 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.
- [Yokoo *et al.*, 1998] 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.