Superstabilizing, Fault-containing Distributed Combinatorial Optimization

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

Abstract

Self stabilization in distributed systems is the ability of a system to respond to transient failures by eventually reaching a legal state, and maintaining it afterwards. This makes such systems particularly interesting because they can tolerate faults, and are able to cope with dynamic environments.

We propose the first self stabilizing mechanism for *multiagent combinatorial optimization*, which works on general networks and stabilizes in a state corresponding to the optimal solution of the optimization problem. Our algorithm is based on dynamic programming, and requires a *linear number of messages* to find the optimal solution in the absence of faults.

We show how our algorithm can be made *super-stabilizing*, in the sense that while transiting from one stable state to the next, our system preserves the assignments from the previous optimal state, until the new optimal solution is found. We offer equal bounds for the stabilization and the superstabilization time.

Furthermore, we describe a general scheme for *fault containment* and fast response time upon low impact failures. Multiple, isolated failures are handled effectively.

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

Introduction

Self stabilization in distributed systems (Dijkstra 1974) is the ability of a system to respond to transient failures by eventually reaching a legal state, and maintaining it afterwards. This property is useful in error-prone distributed systems like distributed sensor networks, or in dynamic environments like control systems or distributed scheduling, where convergence to legal states is ensured without user intervention.

In general, self-stabilizing algorithms have been developed for relatively "low-level" tasks: leader election, spanning tree maintenance (e.g. (Collin & Dolev 1994)) and mutual exclusion. A notable exception is the more recent work of (Collin, Dechter, & Katz 1999) for distributed self-stabilizing constraint satisfaction.

There has also been an attempt at constraint optimization using a distributed, self-stabilizing version of branch and bound in (Yahfoufi & Dowaji 1996). This approach has the drawback that it may be necessary to create an exponential number of agents, because they represent processes corresponding to subproblems.

We propose the first practical, self stabilizing mechanism for *multiagent combinatorial optimization*, which stabilizes in a state corresponding to the optimal solution of the problem. Unlike the previous approaches for constraint satisfaction which are backtracking-based, our algorithm is based on dynamic programming, and requires a linear number of messages to find the optimal solution in the absence of faults. The size of the largest message depends on the *width* of the problem graph. This is an extension of the utility propagation mechanism from (Petcu & Faltings 2005).

We show how our algorithm can be made *super-stabilizing* (Dolev & Herman 1997), in the sense that while transiting from one stable state to the next, the old assignments from the previous optimal state are preserved (similar to a "last-known-good" state), until the new optimal solution is found (without "random" changes to the variables). Furthermore, we describe a general scheme for *fault containment* and fast response time upon low impact failures. Multiple, isolated failures are handled effectively.

Finally, we present experimental results on distributed meeting scheduling problems.

Definitions & notation

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

- $\mathcal{X} = \{X_1, ..., X_m\}$ is the set of variables/agents;
- $\mathcal{D} = \{d_1, ..., d_m\}$ is a set of finite domains of the variables; we can assume equal sizes of the domains;
- $\mathcal{R} = \{r_1, ..., r_p\}$ is a set of relations, where a relation r_i is a function $d_{i1} \times .. \times d_{ik} \rightarrow \Re^+$ which denotes how much utility is assigned to each possible combination of values of the involved variables;

In this paper we consider unary and binary relations, being well-known that higher arity relations can also be expressed in these terms with little modifications. In a MCOP,

*We thank Michael Schumacher and the reviewers for their useful comments. This work has been funded by the Swiss National Science Foundation under contract No. 200020-103421/1.

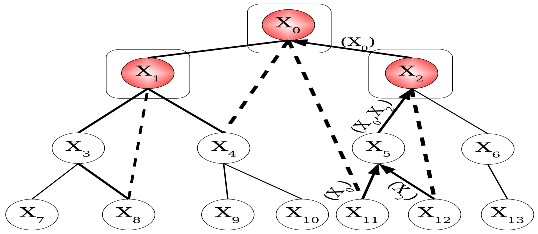

Figure 1: *A problem graph and a rooted DFS tree.*

any value combination is allowed; the goal is to find an assignment $\mathcal{X}^*$ for the variables X_i that maximizes the aggregate utility, i.e. the sum of utilities of individual relations. For a node X_i, we define R_i^j = the relation(s) between X_i and its neighbor X_j.

Pseudotrees

Our method works with a pseudotree arrangement of the problem graph (this is possible for any graph).

Definition 1 *A pseudo-tree arrangement of a graph G is a rooted tree with the same nodes as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g. X_0 and X_{11} in Figure 1).*

As it is already known, a DFS (depth-first search) tree is also a pseudotree, although the inverse does not always hold. We thus use as pseudotree a DFS tree generated by a self-stabilizing DFS algorithm as (Collin & Dolev 1994).

Figure 1 shows an example of a pseudotree that we shall refer to in the rest of this paper. We distinguish between *tree edges*, shown as solid lines (e.g. $8-3$), and *back edges*, shown as dashed lines, that are not part of the spanning tree (e.g. $8-1$, $12-2$, $4-0$). 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. Since our arrangement is a pseudotree, such a tree path is always included in a branch of the tree. For each back-edge, we call the higher node involved in that back-edge its *handler* (e.g. 0, 1 and 2), and the lower node its *initiator* (e.g. 8,4,11,12).

Definition 2 *The parent **P(**X**)** of a node X is the single node on a higher level of the pseudotree that is connected to the node X directly through a tree edge (e.g. $P(X_4) = X_1$). The children **C(**X**)** of a node X are the nodes lower in the pseudotree that are connected to the node X directly through tree edges (e.g. $C(X_1) = \{X_3, X_4\}$). The pseudo-parents **PP(**X**)** of a node X are the nodes higher in the pseudotree that are connected to the node X directly through back-edges ($PP(X_8) = \{X_1\}$). The pseudo-children **PC(**X**)** of a node X are the nodes lower in the pseudotree that are connected to the node X directly through back-edges (e.g. $PC(X_0) = \{X_4, X_{11}\}$).*

SDPOP: a self-stabilizing protocol for MCOP

In a stable state, the system must satisfy the following *legitimacy predicate*: all variables are assigned values that maximize the aggregate utility. Our method is composed of 3 concurrent self-stabilizing protocols:

- self-stabilizing protocol for DFS tree generation: its goal is to create and maintain (even upon faults/topology changes) a DFS tree maintained in a distributed fashion
- self-stabilizing protocol for propagation of utility messages: bottom-up utility propagation along the DFS tree
- self-stabilizing protocol for propagation of value assignments: based on the utility information obtained during the previous protocol, each node picks its optimal value and informs its children (top-down along the DFS tree).

The *SDPOP* algorithm is described in Algorithm 1. The three protocols are initialized and then run concurrently. The following subsections explain in detail the functioning of each of the three subprotocols.

Algorithm 1: *SDPOP - Self-stabilizing distributed pseudotree optimization procedure for general networks.*

SDPOP($\mathcal{X}, \mathcal{D}, \mathcal{R}$**)**: each agent X_i does:

Self-stabilizing DFS protocol: run continuously
if changes in topology, reactivate
after stabilization, X_i knows $P(i), PP(i), C(i), PC(i)$

***UTIL* propagation protocol**: run continuously
get and store all new *UTIL* messages $(X_k, UTIL_k^i)$
if P(i), PP(i), C(i), PC(i), $UTIL_k^i$ or R_i^k changed **then**
 $UTIL_{X_i}^{P(i)} =$
 $\left(\left(\bigoplus_{c \in C(i)} UTIL_c^i\right) \oplus \left(\bigoplus_{c \in \{P(i) \cup PP(i)\}} R_i^c\right)\right) \perp_{X_i}$
 Store $UTIL_{X_i}^{P(i)}$ and send it to P(i)

***VALUE* propagation protocol**: run continuously
get and store all new *VALUE* messages $(X_k, v(X_k))$
if changes in $v(P(i)), v(PP(i))$ or $UTIL_{X_i}^{P(i)}$ **then**
 $v_i^* \leftarrow argmax_{X_i}\left(UTIL_{X_i}^{P(i)}[v(P(i)), v(PP(i))]\right)$
 Send $VALUE(X_i, v_i^*)$ to all $C(i)$ and $PC(i)$

Self-stabilizing DFS tree generation

This protocol has as a goal to establish and maintain a depth-first search tree in a distributed fashion. We use the self-stabilizing DFS algorithm from (Collin & Dolev 1994). In this algorithm, each node maintains a public register with its shortest path from the root. Reading the neighbors' registers and comparing them allows each node to classify its links as tree edges, forward edges and back edges, depending on its own point of view. The node X_i who labeled an edge as a forward edge is its *handler*, and the *pseudoparent* of the other node. The other node X_j involved in that non-tree edge is its *initiator*, and the *pseudochild* of X_i.

Apart from its initial execution, this protocol reactivates whenever any node detects a change in the problem topology (addition/removal of variables or relations).

Self-stabilizing *UTIL* propagation

This protocol reactivates whenever it detects a change either in the previous protocol (DFS generation, meaning that the topology of the problem has changed), or in the valuation structure of the optimization problem (values are added/removed, valuations of tuples change in relations).

The *UTIL* propagation starts bottom-up from the leaves and propagates upwards only through tree edges. The agents send *UTIL* messages to their parents. Intuitively, such a message informs a parent node X_j how much utility $u^*_{X_i}(v^k_j)$ each one of its values v^k_j gives in the optimal solution of the whole subtree rooted at the sending child, X_i. If there is no back-edge connecting a node from X_i's subtree to a node above X_j, then these valuations depend only on X_j's values, and the message from X_i to X_j is a vector with $|dom(X_j)|$ values. Otherwise, these back-edges have to be taken into account, and their handlers are present as *dimensions* in the message from X_i to X_j.

Definition 3 *$UTIL^j_i$, the UTIL message sent by agent X_i to agent X_j is a multidimensional matrix, with one dimension for each variable present in the context. $dim(UTIL^j_i)$ is the set of individual variables in the message. Note that always $X_j \in dim(UTIL^j_i)$.*

The semantics of such a message is similar to an n-ary relation having as scope the variables in the context of this message (its *dimensions*). The size of such a message is the product of the domain sizes of the variables from the context.

Definition 4 *The $\oplus$ operator (join): $UTIL^j_i \oplus UTIL^j_k$ is the join of two UTIL matrices. This is also a matrix with $dim(UTIL^j_i) \cup dim(UTIL^j_k)$ as dimensions. The value of each cell in the join is the sum of the corresponding cells in the two source matrices.*

Example: given 2 matrices $UTIL^j_i$ and $UTIL^j_k$, with $dim(UTIL^j_i) = \{X_1, X_j\}$ and $dim(UTIL^j_k) = \{X_2, X_j\}$, then the value corresponding to $\langle X_1 = v^p_1, X_2 = v^q_2, X_j = v^r_j\rangle$ is $UTIL^j_i(X_1 = v^p_1, X_j = v^r_j) + UTIL^j_k(X_2 = v^q_2, X_j = v^r_j)$. Also, $dim(UTIL^j_i \oplus UTIL^j_k) = \{X_1, X_2, X_j\}$.

Definition 5 *The $\perp$ operator (projection): if $X_k \in dim(UTIL^j_i)$, $UTIL^j_i \perp_{X_k}$ is the projection through optimization of the $UTIL^j_i$ matrix along the X_k axis: for each tuple of variables in $\{dim(UTIL^j_i) \setminus X_k\}$, all the corresponding values from $UTIL^j_i$ (one for each value of X_k) are tried, and the best one is chosen. The result is a matrix with one less dimension (X_k).*

Notice that a relation R^j_i (between X_i and X_j), is just a special case of *UTIL* matrix, with 2 dimensions i and j. Therefore, operators $\oplus$ and $\perp$ apply to it as well.

Example 1: for a relation R^j_i, $R^j_i \perp_{X_i}$ is a vector $UTIL^j_i$ containing the best utilities for each value of X_j, when the corresponding optimal value of X_i is chosen. Example 2: for a vector $UTIL^j_i$, $UTIL^j_i \perp_{X_j}$ is the optimal value of X_j. Example 3: in Figure 1, X_4 computes its $UTIL^1_4$ message for X_1 (see Equation 1, and Table 1 for an extended form):

$X_4 \rightarrow X_1$	$X_1 = v^0_1$	...	$X_1 = v^{m-1}_1$
$X_0 = v^0_0$	$u^*_{X_4}(v^0_0, v^0_1)$	...	$u^*_{X_4}(v^0_0, v^{m-1}_1)$
...	...	...	...
$X_0 = v^{n-1}_0$	$u^*_{X_4}(v^{n-1}_0, v^0_1)$	...	$u^*_{X_4}(v^{n-1}_0, v^{m-1}_1)$

Table 1: *UTIL* message sent from X_4 to X_1, in Figure 1

$$UTIL^1_4 = \underbrace{(\overbrace{\underbrace{\overbrace{UTIL^4_9 \oplus UTIL^4_{10}}^{dim=\{X_4\}} \oplus R^0_4}_{dim=\{X_4,X_0\}} \oplus R^1_4}^{dim=\{X_4,X_0,X_1\}}) \perp_{X_4}}_{dim=\{X_0,X_1\}} \quad (1)$$

The leaf nodes initiate the process (e.g. $UTIL^3_7 = R^3_7 \perp_{X_7}$). Then each node X_i relays these messages according to the following process:

- Wait for *UTIL* messages from all children. Since all the respective subtrees are disjoint, joining messages from all children gives X_i exact information about how much utility each of its values yields for the whole subtree rooted at itself. In order to assemble a similar message for its parent X_j, X_i has to take into account R^j_i and any back-edge relation it may have with nodes above X_j. Performing the join with these relations and projecting itself out of the result (see line 10 in Algorithm 1) gives a matrix with all the optimal utilities that can be achieved for each possible combination of values of X_j and the possible context variables. Thus, X_i can send to X_j its $UTIL^j_i$ message (see Equation 1, and Table 1 for $UTIL^1_4$).
- If root node, X_i receives all its *UTIL* messages as vectors with a single dimension, itself. It can then compute the optimal overall utility corresponding to each one of its values (by joining all the incoming *UTIL* messages) and pick the optimal value for itself (project itself out).

The back-edge handlers are present as extra dimensions in the *UTIL* messages that travel through the system along the tree-path associated with the respective back-edge. Example: X_3 gets $UTIL^3_8$ from X_8, with $dim(UTIL^3_8) = \{X_3, X_1\}$. X_3 joins this message with $UTIL^3_7$ and R^1_3 and projects itself out, in order to compute the message for its parent: $UTIL^1_3 = \left(UTIL^3_8 \oplus UTIL^3_7 \oplus R^1_3\right) \perp_{X_3}$. $dim(UTIL^1_3) = \{X_1\}$. When $UTIL^1_3$ reaches X_1, it will be joined with $UTIL^1_4$ ($dim(UTIL^1_4) = \{X_1, X_0\}$), and X_1 will project itself out, to obtain $UTIL^0_1$. Thus, the propagation of X_1 as a dimension in the *UTIL* messages starts from X_8 (initiator of R^1_8) to X_3 and ends at X_1(handler).

Self-stabilizing *VALUE* propagation

The root of the pseudotree initiates the top-down *VALUE* propagation phase by sending a *VALUE* message to its children and pseudochildren, informing them about its chosen value. Then, each node X_i is able to pick the optimal value for itself upon receiving all *VALUE* messages from its parent and pseudoparents. This is the value which was determined in X_i's *UTIL* computation to be optimal for this particular instantiation of the parent/pseudoparents variables. X_i then passes its value on to its children and pseudochildren. Thus, there is exactly one *VALUE* message for each edge, totalling to $|\mathcal{R}|$ *VALUE* messages that travel from the root to the leaves.

Algorithm complexity

By construction, in the absence of faults, the number of messages our algorithm produces is linear: there are $n-1$ *UTIL* messages - one through each tree-edge (n is the number of nodes in the problem), and m *VALUE* messages - one through each edge (m is the number of edges). The DFS construction also produces a linear number of messages (good algorithms require $2 \times m$ messages).

The complexity of this algorithm lies in the size of the *UTIL* messages (the *VALUE* messages have constant size).

Theorem 1 *The largest UTIL message produced by Algorithm 1 is space-exponential in the width of the pseudotree induced by the DFS ordering used.*

PROOF. Dechter ((Dechter 2003), chapter 4, pages 86-88) describes *the fill-up method* for obtaining the *induced width*. First, we build the *induced graph*: we take the DFS traversal of the pseudotree as an ordering of the graph and process the nodes recursively (bottom up) along this 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.

It is shown in (Dechter 2003) that the width of a tree (no back-edges) is 1. Actually the back-edges are the ones that influence the width: a single backedge produces an induced width of 2. From the construction of the induced tree, it is easy to see that several backedges produce increases in the width only when their tree-paths overlap on at least one edge, and their respective handlers are different; otherwise their effects on the width do not combine. Thus, the width is given by the size of the maximal set of back-edges which have overlapping tree-paths and distinct handlers.

During the *UTIL* propagation, the message size varies; the largest message is the one with the most dimensions. We have seen that a dimension X_i is added to a message when a back-edge with X_i as a handler is first encountered in the propagation, and travels through the tree-path associated with the back-edge. It is then eliminated by projection when the message arrives at X_i. The maximal dimensionality is therefore given by the maximal number of overlaps of tree-paths associated with back-edges with distinct handlers.

We have shown that both the induced width and the maximal dimensionality are equal to the same amount. □

In problems with high induced width, the *UTIL* messages can be big. If this is the case, they can be "compressed" by sending a mixture of *UTIL* messages and unprocessed relations. This saves bandwidth by reducing the maximal message size from $|dom|^w$ to a sum of $|dom|^{max}$ (max is a bound on maximal dimensionality). However, the computational effort remains the same.

Self stabilization of SDPOP

Theorem 2 *SDPOP is self-stabilizing: even upon transient perturbations/failures, it will always reach a stable state where all variables have the assignments corresponding to the optimal solution of the optimization problem.*

PROOF. We use a chaining technique and the fair composition principle (Dolev 2000) to prove the self-stabilization of SDPOP. Firstly, the self-stabilizing DFS algorithm is guaranteed to eventually build a valid DFS tree if no more changes are made to the topology of the problem.

Thus, the utility propagation will eventually start with a correct DFS tree. By design, this protocol reaches after at most $n-1$ messages a stable state where all the nodes have correct *UTIL* messages from all their children (if there are no more changes in topology or valuation structure).

Thirdly, the *VALUE* propagation protocol is guaranteed to finally start from a stable state, where each node has correct *UTIL* information. Based on that, this protocol reaches after at most $|\mathcal{R}|$ *VALUE* messages a stable state where all variables are assigned their optimal values. □

Theorem 3 *Upon single faults, SDPOP stabilizes after at most k UTIL messsages and at most $|\mathcal{R}|$ VALUE messages (k is the length of the longest branch in the pseudotree). In a synchronous implementation, stabilization is reached in at most $2 \times k$ steps.*

PROOF. By construction, the *UTIL* propagation initiated by any node travels only bottom-up towards the root; therefore, in the worst case, when a fault occurs at the leaf which is farthermost from the root, there are as many *UTIL* messages as nodes on that longest branch. Furthermore, in the worst case, where the fault changes *every* value assignment, there occurs a full-blown *VALUE* propagation of $|\mathcal{R}|$ linear messages. In the synchronous implementation, there are at most k steps for bottom-up *UTIL* propagation and at most k steps for top-down *VALUE* assignments. □

Experimental evaluation

We experimented with distributed meeting scheduling in an organization with a hierarchical structure (a tree with departments as nodes, and a set of agents working in each department). The CSP model is the PEAV model from (Maheswaran *et al.* 2004). Each agent has multiple variables: one for the start time of each meeting it participates in, with 8 timeslots as values. Mutual exclusion constraints are imposed on the variables of an agent, and equality constraints are imposed on the corresponding variables of all agents involved in the same meeting. Private, unary constraints placed by an agent on its own variables show how much it values each meeting/start time. Random meetings are generated, each with a certain utility for each agent. The objective is to find the schedule that maximizes the overall utility.

Table 2 shows how our algorithm scales up with the size of the problems. All experiments are run on a 1.6Ghz laptop. Notice that the total number of messages includes the

Agents	30	40	70	100	200
Meetings	14	15	34	50	101
Variables	44	50	112	160	270
Constraints	52	60	156	214	341
Messages	95	109	267	373	610
Max message size	512	4096	32k	256k	256k
Solving time (s)	2.1	7.2	21.6	43.4	72.3
Δ-changes	5	5	12	16	27
Δ-repair-steps	15	16	35	43	48

Table 2: SDPOP tests on meeting scheduling.

VALUE messages (constant size), and that due to the fact that intra-agent subproblems are denser than the rest of the problem, high-dimensional messages are likely to be virtual, intra-agent messages (not actually transmitted). To our knowledge, these are by far the largest optimization problems solved with a complete, distributed algorithm (200 agents, 101 meetings, 270 variables, 341 constraints). Previously, (Maheswaran *et al.* 2004) reported on experiments with 33 agents, 12 meetings, 47 variables, 123 constraints. The algorithm used there is *ADOPT*, which is not a self-stabilizing algorithm.

Additionally, once the solutions are found, we apply simultaneous perturbations amounting to 10% of the agents, to simulate change of preferences. Δ-changes shows how many preferences changed, and Δ-repair-steps shows how many synchronous steps are required for stabilization in the new optimal solution. To our knowledge there are no other results on self-stabilizing distributed optimization as yet.

Protocol Extensions

Self stabilizing algorithms generally do not provide any guarantees about the way the system transits from a valid state to the next, upon perturbations. Superstabilization and fault containment are two features addressing this issue.

Super-stabilization

Super-stabilization is a guarantee that the protocol satisfies a *passage predicate* at all times, transitional states included (Dolev 2000). Typically, this is a safety property, weaker than the legitimacy predicate, but nevertheless useful.

Assuming that the occasional perturbations of the system are not so drastic that they completely change the old solution, we define the passage predicate as maintaining the previous optimal assignment while the new one is recomputed. This aspect can be vital (e.g. while controlling an industrial process in real-time, random settings applied to various installations during the search for the optimal solution can be dangerous). This poses a problem for backtracking algorithms, since they produce "random" variable assignments in their search for the optimal solution, as instantiations are made in order to try them out and compute their costs. Keeping this predicate true in transitional states thus requires extra effort.

In contrast, this "stability" is very natural to our algorithm, since first all the *UTIL* information is propagated and then the value assignment phase begins, with already stable/optimal values. This requirement is briefly broken by *SDPOP* after the new stabilization of the *UTIL* protocol, where the *VALUE* propagation begins. Typically, this is a short process, since a linear number of linear size messages is used. Complete atomicity of the switch to the new solution is also possible, provided the messages are transmitted synchronously. The *VALUE* propagation proceeds as before, but the nodes change their value only after a number of clock ticks, not immediately as before. The number of ticks is given for each node as the difference between the length of the longest branch in the pseudotree and its level in the pseudotree (this is easy to obtain from the DFS protocol). This ensures that the switch to the new optimum happens atomically, when the *VALUE* propagation reached all leaves. Notice that the superstabilization time is the same as in normal *SDPOP*, just that the assignments are made all at the end.

Fault-containment

Other aspects of self-stabilization are the quick response time in case of "minor" changes and the containment of their effects to confined areas in their vicinity (Ghosh *et al.* 1996).

Fault-containment in the DFS construction It is obvious that changes in the DFS structure will adversely affect the performance of our algorithm, since some of the *UTIL* messages will have to be recomputed and retransmitted. Therefore, it is desirable to maintain as much as possible the current DFS tree. Describing such a protocol is beyond the scope of this paper. We use techniques similar to (Dolev & Herman 1997; Ghosh *et al.* 1996).

Fault-containment in the *UTIL/VALUE* protocols In the previous *UTIL* protocol, upon a perturbation all *UTIL* messages on the tree-path from the fault to the root are recomputed and retransmitted. This is sometimes wasteful, since some of the faults have limited, localized effects, which need not propagate through the whole problem. To limit this, we change the *UTIL* propagation in two respects.

Firstly, when a change occurs, and an *UTIL* message needs to be retransmitted, it is compared to the one which was previously sent; in case there are no differences, it is simply discarded. Thus, the influences of a change in terms of utility variations diminish from one hop to the next, until the propagation stops altogether.

Secondly, we *rescale* all *UTIL* matrices by subtracting from each element the lowest utility value present in that matrix. This is a sound operation because in such a propagation algorithm the relative differences in valuation are important, and not the absolute valuations. Intuitively, if a node X_i has 3 values, then receiving 0,1 and 2 as valuations for these values is no different than receiving 10,11 and 12. This makes more irrelevant changes not trigger a propagation anymore.

Similarly, *VALUE* messages propagate only as long as there is a change in assignment performed; thus, low magnitude changes in the problem are likely to even go unnoticed by nodes which are relatively far away.

Fast response time upon low-impact faults In any real-time system, optimal decisions have to be made as quickly as possible. In some cases, we want to respond to a perturbation by *immediately* assigning the new optimal value to the "touched" variable, and then gradually re-assigning the neighboring ones to their new optimal values, until all the system is again stabilized. For example, when a truck breaks down, we want to immediately re-route the closest one to take its load, and then gradually re-route the other trucks to the new optimum. We also want to deal effectively with multiple simultaneous faults which are unrelated (their effects are localized in different parts of the problem).

To achieve this, each node needs global utility information. Then it is easy to immediately assess *locally* the *global* effect of a perturbation on any node. In the previous protocol, the root had global information, but all other nodes had accurate *UTIL* information only about their subtrees. We extend the *UTIL* propagation by making it *uniform*: now it also goes top-down, from each node to its children. A message from a parent to its child summarizes the utility information from all the problem except the subtree of that child. Joining this message with the ones received from its children gives each node a global view of the system, logically making each node in the system equivalent to the root.

The process is initiated by the root. Each X_i (root included) computes for each of its children X_j a $UTIL_i^j$ message. X_i first builds the join: $JOIN_i^j = R_i^j \oplus \left(\bigoplus_{c \in \{TN(i) \setminus X_j\}} UTIL_c^i\right)$ ($TN(i)$ is the set of *tree-neighbors* of X_i). e.g.: $JOIN_2^5 = R_2^5 \oplus UTIL_0^2 \oplus UTIL_6^2$).

Then, appropriate projections have to be applied, and the message is sent to the child. Intuitively, $UTIL_i^j$ ($X_i \rightarrow X_j$) has to *match* the dimensions of $UTIL_j^i$ ($X_j \rightarrow X_i$), except that X_j has to be added (taken care of by the join of R_i^j) and X_i may need to be projected out (unless there is any back-edge connecting X_i with a node in X_j's subtree). When the DFS algorithm from (Collin & Dolev 1994) is used, it is possible for a node X_i to determine which is the tree-path associated with each one if its back-edges by comparing the suffix/prefix of the root-paths of its neighbors with their ids. If there is no back-edge R_i^k s.t. its associated tree-path goes through X_j, then X_i projects itself out of the brute message; otherwise not. Once X_i has determined the relevant dimensions, it projects out everything else:

$$UTIL_i^j = JOIN_i^j \perp_{X_k \in \{dim(JOIN_i^j) \setminus dim(UTIL_i^j)\}}$$

Examples: $dim(UTIL_0^2) = \{X_0, X_2\}$, $dim(UTIL_2^5) = \{X_0, X_2, X_5\}$, $dim(UTIL_5^{11}) = \{X_0, X_{11}\}$. When computing $UTIL_0^2$, X_0 sees that the tree-path of R_{11}^0 goes through X_2, therefore, it does not project itself out of $JOIN_0^2$. Similarly, X_2 keeps itself in $UTIL_2^5$, but projects itself out of $JOIN_2^6$; $UTIL_5^{11} = JOIN_5^{11} \perp_{\{X_5, X_2\}}$.

Upon a change a node can now immediately *locally* compute its new *globally* optimal value. In case the perturbation implies a change in utility for several other variables, the propagation spreads, but only as far as necessary. Thus, low impact perturbations require just a few messages to reach the new optimal state. In case their impact areas do not overlap, they are effectively dealt with: in the best case, n simultaneous perturbations are dealt with in $O(1)$ time. Obviously, in the worst case the propagation spreads to all nodes.

Concluding Remarks

We propose the first self stabilizing mechanism for *distributed combinatorial optimization*, which works for general constraint networks and stabilizes in an optimal solution of the optimization problem. We offer equal bounds for the stabilization and the superstabilization time.

Closest in spirit with our work is the self stabilizing constraint satisfaction approach from (Collin, Dechter, & Katz 1999). Our contributions beyond this work are: first, we extend the framework for optimization, not just satisfaction. Second, our algorithm is based on dynamic programming and requires a linear number of messages to find the optimal solution in the absence of faults. Our algorithm is thus well suited for distributed systems, where many small messages produce big overheads. Third, we presented interesting extensions of the basic algorithm, achieving superstabilization, fault-containment and fast response time.

The contributions beyond the protocol from (Petcu & Faltings 2005) are manyfold: self stabilization, superstabilization, fault containment, uniform utility propagation.

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

Future work includes application to several problem domains and tuning the fault-containment scheme to common kinds of failures.

References

Collin, Z., and Dolev, S. 1994. Self-stabilizing depth-first search. *Information Processing Letters* 49(6):297–301.

Collin, Z.; Dechter, R.; and Katz, S. 1999. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*.

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

Dijkstra, E. W. 1974. Self stabilizing systems in spite of distributed control. *Communication of the ACM* 17(11):643–644.

Dolev, S., and Herman, T. 1997. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*.

Dolev, S. 2000. *Self-Stabilization*. MIT Press.

Ghosh, S.; Gupta, A.; Herman, T.; and Pemmaraju, S. V. 1996. Fault-containing self-stabilizing algorithms. In *Symposium on Principles of Distributed Computing*, 45–54.

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

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

Yahfoufi, N., and Dowaji, S. 1996. A self-stabilizing distributed branch-and-bound algorithm. In *Computers and Communications*, 246–252.