Optimal Solution Stability in Continuous-Time Optimization

Adrian Petcu and Boi Faltings

Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne (Switzerland)
{adrian.petcu, boi.faltings}@epfl.ch

Abstract. We define the *distributed, continuous-time combinatorial optimization problem*. We propose a general, semantically well-defined notion of *solution stability* in such systems, based on the *cost of change* from an already implemented solution to the new one. This approach allows maximum flexibility in specifying these costs through the use of *stability constraints*. We present the first mechanism for combinatorial optimization that guarantees optimal solution stability in dynamic environments, based on this notion of solution stability.
In contrast to current approaches which solve sequences of static CSPs, our mechanism has a lot more flexibility by allowing for a much *finer-grained vision of time*: each variable of interest can be assigned and reassigned *its own commitment deadlines*, allowing for a *continuous-time* optimization process. We emphasize that this algorithm deals with *dynamic problems*, where variables and constraints can be added/deleted at runtime.
We show the efficiency of this approach with experimental results from the distributed meeting scheduling domain. We describe here a distributed algorithm, but it can easily be centralized.

Keywords:distributed AI, dynamic combinatorial optimization, solution stability

1 Introduction

Real time systems are increasingly important, with applications ranging from entertainment software (games, multimedia) to mission critical process control like automotive/aircraft control, nuclear reactor control, etc. There is no universally accepted definition of real-time systems. A seemingly popular one is this: a real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. We add that it is important for the users of such a system to be able to specify deadlines until which the tasks the system is carrying out must be achieved.

Dynamic systems often have a certain notion of *inertia*: once in a stable state, a system cannot adapt instantaneously to changes in the environment, but only after a certain time, and with a certain "effort". Constraint optimization in a dynamic environment [2] is no exception: it takes time for a system to find the optimal solution to an optimization problem upon changes in the problem, and there is usually a certain amount of change that has to be inflicted upon the old solution for it to *adapt* to the new situation. Often it

is impossible to change some parts of the old solution, or there is a high cost associated with change. Examples: resources are already consumed, re-routing trucks to serve a new customer costs extra fuel, re-assigning nurses to shifts in a hospital, etc.

Solution stability is an important concept that captures this "change" of solutions in dynamic systems, and tries to minimize its effects.

Current approaches [8, 9] define solution stability in dynamic CSP with respect to the number of variable assignments that need to be changed in order to reach again a consistent state upon a change in the problem. We argue that the number of assignments that change between solutions is irrelevant; what matters is the total *cost* that is induced by these changes, once the assignments are made. If these costs are outweighed by the benefits in solution quality, one can gain from making as many changes as necessary. In the truck routing example, if rerouting a few trucks means serving a new task that produces a benefit which far exceeds the additional fuel costs, then we reroute the trucks, and gain the benefits.

Furthermore, current approaches have a rather coarse grained vision of time: they assume the world holds still from one execution of the algorithm to the next, allowing them to solve static CSPs in each state, and minimize assignment changes from one solution to the next. This is not adequate in a multiagent system, because it means that all agents and tasks have the same rigid deadline. There are two ways to operate in such a system. First, everybody has to synchronize their deadlines by putting them off until the latest deadline in sight, then set a new deadline for the next wave of tasks, and then another one, etc. This approach clearly lacks flexibility, and a lot of agents sit idle while waiting for a distant deadline of another agent to pass. Second, one creates a lot of intermediate deadlines for the whole set of agents/tasks, each one corresponding to a real deadline of one task. This approach introduces a big synchronization overhead, because the agents have to synchronize their actions at a lot of intermediate points which do not even concern them.

In contrast, we propose a much finer-grained approach, where we model time explicitly, and assign *deadlines* to each variable of interest, thus allowing for a *continuous* solving process, with independent *commitment times* for each variable. It is obvious that our approach can easily be reduced to the "classical" approaches by setting for all variables the same, synchronized deadlines. Like this, the solving process finds a solution, assigns the variables synchronously at the common deadline, and is ready for the next cycle.

In a dynamic environment, optimizing continuously and never providing a solution is obviously not useful. Occasionally, one has to make some commitments, and then adjust them as new events unfold. We identify two kinds of commitments: *soft commitments* and *hard commitments*. *Soft commitments* model contracts with penalties, and can be revised if the benefit extracted from the change outweighs its cost. *Hard commitments* model irreversible processes, and are impossible to undo.

The rest of this paper is structured as follows: section 2 formally describes the optimization problem. Section 3 presents the *SDPOP* algorithm from [7], upon which we will build our present work. Section 4 introduces the continous-time optimization problem and some assumptions we make. Section 5 presents our approach to solution stability. Section 6 presents the *RSDPOP* algorithm, a realtime optimization algorithm

that offers optimal solution stability. Section 7 presents an experimental evaluation. Section 8 presents some considerations for practical implementations of this algorithm. Section 9 concludes.

2 Definitions & Notation

Definition 1. *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/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 denotes how much utility is assigned to each possible combination of values of the involved variables.*

All these sets can vary with time.

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 aggregate overall utility.

3 SDPOP: a self-stabilizing protocol for MCOP

The optimization problems that we are dealing with are dynamically changing over time. For such dynamic environments, self stabilizing algorithms([3]) are particularly well suited, since they guarantee some desired behavior even when there are changes in the problem. Therefore, we use the *SDPOP* algorithm from [7] as a basic building block for this work. This algorithm is a self-stabilizing algorithm and works for dynamic, distributed *MCOP* problems. *SDPOP* guarantees stabilization in the optimal solution of the optimization problem. We present in this section this basic algorithm, and extend it to the RMCOP model in section 6.

In a stable state, *SDPOP* satisfies the following *legitimacy predicate*: all variables are assigned values that maximize the aggregate utility. It 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.

Subsection 3.1 gives some background on pseudotrees and how they can be used in optimization.

The following 3 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 $PC(i)$

3.1 Pseudotrees

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

Definition 2. *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 [1].

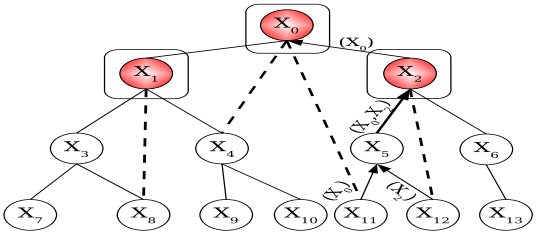

Fig. 1. *A problem graph and a rooted DFS tree.*

In the example of figure 1 one can see that some of the edges of the original graph are not part of the spanning tree (otherwise the problem is 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*. A *tree-path* is a path entirely made of tree edges. A *tree-path associated with a back-edge* is the tree-path connecting the two nodes involved in the back-edge (as our arrangement is a pseudotree, such a tree path is always unique and included in a branch of the tree).

For each back-edge, the higher node is called the *back-edge handler*, and the lower one is its *initiator* (in Figure 1 0, 1, 2 are *handlers*, and 8,4,11,12 are *initiators*).

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

Definition 3. ***P(***X***)*** *- the parent of a node X: the single node on a higher level of the pseudotree that is connected to the node X directly through a tree edge (e.g. $P(X_4) = X_1$).* ***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 pseudotree that are connected to the node X directly through back-edges (e.g. $PC(X_0) = \{X_4, X_{11}\}$).*

3.2 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 [1]. In the terminology of that paper, non-tree edges are labeled as forward edges and back edges, depending on the point of view of the classifying node. 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).

3.3 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 4. *$UTIL_i^j$ - the UTIL message sent by agent X_i to agent X_j; this is a multidimensional matrix, with one dimension for each variable present in the context. $dim(UTIL_i^j)$ - the whole set of dimensions (variables) of the message ($X_j \in dim(UTIL_i^j)$ always).*

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 5. *The $\oplus$ operator (join): $UTIL_i^j \oplus UTIL_k^j$ is the join of two UTIL matrices. This is also a matrix with $dim(UTIL_i^j) \cup dim(UTIL_k^j)$ 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 $UTIL_i^j$ and $UTIL_k^j$, with $dim(UTIL_i^j) = \{X_1, X_j\}$ and $dim(UTIL_k^j) = \{X_2, X_j\}$, then the value corresponding to $\langle X_1 = v_1^p, X_2 = v_2^q, X_j = v_j^r\rangle$ is $UTIL_i^j(X_1 = v_1^p, X_j = v_j^r) + UTIL_k^j(X_2 = v_2^q, X_j = v_j^r)$. Also, $dim(UTIL_i^j \oplus UTIL_k^j) = \{X_1, X_2, X_j\}$.

Definition 6. *Given a multidimensional UTIL matrix H and an instantiated subset D of its dimensions ($D \subset dims(H)$), a **slice** through H along D, $H[D]$ is a lower-dimensionality matrix S that has as dimensions $\{d | d \in \{dims(H) \setminus D\}\}$ and as values the values from H that correspond to the tuples $\{dims(H) \setminus D\}$. If $D = dims(H)$, $H[D]$ is the corresponding value of H.*

Example: in the message $UTIL_4^1$ from Table 1, a slice $UTIL_4^1[X_0 = v_0^0]$ along $X_0 = v_0^0$ is the first row of the table.

Definition 7. *The $\perp$ operator (projection): if $X_k \in dim(UTIL_i^j)$, $UTIL_i^j \perp_{X_k}$ is the projection through optimization of the $UTIL_i^j$ matrix along the X_k axis: for each tuple of variables in $\{dim(UTIL_i^j) \setminus X_k\}$, all the corresponding values from $UTIL_i^j$ (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_i^j (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_i^j, $R_i^j \perp_{X_i}$ is a vector $UTIL_i^j$ 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 $UTIL_i^j$, $UTIL_i^j \perp_{X_j}$ is the optimal value of X_j. Example 3: in figure 1, X_4 computes its $UTIL_4^1$ message for X_1 (see equation 1, and table 1 for an extended form):

$$UTIL_4^1 = \underbrace{(\overbrace{\underbrace{UTIL_9^4 \oplus UTIL_{10}^4}_{dim=\{X_4\}} \oplus R_4^0}^{dim=\{X_4,X_0\}} \oplus R_4^1) \perp_{X_4}}_{dim=\{X_0,X_1\}} \overbrace{\qquad\qquad}^{dim=\{X_4,X_0,X_1\}} \tag{1}$$

$X_4 \rightarrow X_1$	$X_1 = v_1^0$	$X_1 = v_1^1$	...	$X_1 = v_1^{m-1}$
$X_0 = v_0^0$	$u^*_{X_4}(v_0^0)$	$u^*_{X_4}(v_0^0)$	...	$u^*_{X_4}(v_0^0)$
...	...	...	...	...
$X_0 = v_0^{n-1}$	$u^*_{X_4}(v_0^{n-1})$	$u^*_{X_4}(v_0^{n-1})$	...	$u^*_{X_4}(v_0^{n-1})$

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

The leaf nodes initiate the process (e.g. $UTIL_7^3 = R_7^3 \perp_{X_7}$). 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_i^j 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 $UTIL_i^j$ message (see equation 1, and table 1 for $UTIL_4^1$).
- 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).

3.4 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 of 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, $edges$ *VALUE* messages travel from the root to the leaves throughout the graph.

3.5 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 linear 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. This has been proved in [6] and [7]. A proof sketch is that both the maximal dimensionality and the induced width are equal to the maximal number of overlaps of tree-paths associated with back-edges with distinct handlers.

□

3.6 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. This has been proved in [7]. In brief, the idea is that the DFS protocol eventually stabilizes, then the *UTIL* protocol, and finally the *VALUE* protocol. □

Theorem 3. *Upon single faults, SDPOP stabilizes after at most k UTIL messsages and at most $edges$ 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 $|edges|$ 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. □

4 Continuous time optimization

In dynamic systems, changes occur all the time, and optimization is a continuous process. In some cases, it is required to decide on the values of at least a subset of the variables of the problem, and fix them to some optimal values. A simple example is a dynamic scheduling problem, where at some point one has to fix some tasks, and start working on them, otherwise the system would accomplish nothing but endless scheduling. Time has to be modeled explicitly, and taken into account.

In our approach, upon defining the optimization problem, the designer has the opportunity to specify *commitment deadlines* for each variable: deadlines until which a value must be assigned to the respective variable.

Thus, depending on their semantics, some of the variables in the problem may be assigned such deadlines. Then, at the expiration of the deadline, these variables commit to their current optimal values. In case there is no need for a variable to be fixed at a given point in time, one can simply not assign any deadline to it, leaving it in a "floating" state, where it can always be freely assigned to its (current) optimal value.

We identify two kinds of commitments:

1. *Soft commitments* model contracts with penalties, and can be revised if the benefit extracted from the change outweighs its cost. Examples: a truck can be re-routed to a longer route if it picks up an additional package along the way, which yields an increase in revenue. A supplier may be willing to pay penalties for not delivering on time some goods to client X, if he gets a better revenue by serving customer Y first, etc. This kind of commitment can be modeled with *stability constraints* (see definition 8). We emphasize that deadlines for soft commitment can be re-specified at the expiration of the original ones. In fact, any variable in the system can have assigned to it a sequence of soft-commitment deadlines, possibly followed by a single hard-commitment deadline.
2. *Hard commitments* model irreversible processes, and are impossible to undo (example: production of good *X* already started, and resource *Y* was already consumed). When a hard commit is done, the variable is assigned to its committed value, it is marked "dead" and is logically eliminated from the problem. Physical elimination can be either immediate, or postponed for efficiency reasons, to be removed simultaneously with other *dead* variables. One can design a *garbage collection policy* to deal with dead variables. Hard commitments can be modelled as stability constraints with very large costs.

Definition 8. *A **stability constraint** σ_i is a pseudo-binary constraint on X_i. The semantics of such a constraint is simple: if X_i is assigned to v_i^1, then $\sigma_i(v_i^1 \rightarrow v_i^2)$ denotes how much it costs to change X_i's value to v_i^2. Please refer to Table 2 for an example stability constraint on X_i.*

σ_i	$X_i = v_i^0$	$X_i = v_i^1$	...	$X_i = v_i^{m-1}$
$X_i = v_i^0$	0	$\sigma_i(v_i^0 \rightarrow v_i^1)$	...	$\sigma_i(v_i^0 \rightarrow v_i^{m-1})$
...	...	...	...	...
$X_i = v_i^{m-1}$	$\sigma_i(v_i^{m-1} \rightarrow v_i^0)$	$\sigma_i(v_i^{m-1} \rightarrow v_i^1)$	...	0

Table 2. Stability constraint on X_i

Definition 9. *Formally, a discrete **realtime multiagent constraint optimization problem** (RMCOP) is a tuple $< \mathcal{X}, \mathcal{D}, \mathcal{R}, \mathcal{S}, \mathcal{T} >$ that extends the MCOP definition with:*

- *$\mathcal{S} = \{\sigma_1, ..., \sigma_m\}$ is a set of stability constraints*
- *$\mathcal{T} = \{t_1, ..., t_m\}$ is a set of commitment deadlines: times until the corresponding variable has to commit to a value. Deadlines can be for hard or soft commitments.*

4.1 Assumptions

Distributed real-time computing systems can be categorized as *asynchronous* and *synchronous*. The fully synchronous model assumes that each process has a bounded time between its execution steps, messages are transmitted and received in bounded time, and local clocks are synchronized. The fully asynchronous model does not require synchronized local clocks, and allows for arbitrary times for message delivery and between execution steps. The asynchronous model is obviously more general and flexible, but it is also less useful because a number of important properties (such as distributed

agreement) have been proven impossible even under very weak conditions in the asynchronous model, but are readily achievable in the synchronous model.

For our purposes, it is sufficient to have synchronized clocks, which is weaker than a fully synchronous system. This is not required for the optimization algorithm itself, but for the part that deals with the commitment deadlines.

It is obvious that optimality requires the deadlines imposed on the variables to have reasonable values, at least as great as the solving time. Otherwise, no system, centralized or distributed, can guarantee optimal solutions. Usually, this is not a problem, since the solving time is orders of magnitude smaller than the time scale of the process being optimized. Typical examples include vehicle routing, nurse scheduling, assembly lines, meeting scheduling, etc.

When this requirement is not possible to achieve, and the deadlines are very tight, the solution could be an anytime algorithm like AnyPOP from [5]. Such an algorithm provides increasingly accurate solutions even before completion. The best solution at the deadline is chosen.

5 Solution Stability

Current approaches define solution stability in dynamic CSP with respect to the number of variable assignments that need to be changed in order to reach again a consistent state upon a change in the problem. There are two approaches to achieve this kind of stability: first approach [8] is a curative approach: once a change occurs in the problem, they seek the new solution which is closest to the previous one, thus requiring a minimal number of changes. The second approach [9] is a proactive approach: when generating a solution in the first place, one tries to find robust solutions, which are likely to remain valid even upon changes in the problem, thus requiring little or no adjustment.

We break away from this definition of stability by looking at the process from a *cost* perspective. We argue that actually the number of assignments that change is irrelevant; what matters is the total *cost* that is induced by these changes, once the assignments are made.

To see this, consider a scheduling example in which a company makes a schedule for a lot of small interconnected tasks that lead to the assembly of a product which is supposed to be delivered at a specified time. The manufacturing process starts, but there is a change (e.g. a machine breaks down). Keeping the old schedule valid as much as possible (as it happens when stability is defined as minimizing assignment changes) may lead to the final delivery date being pushed back for 2 months, thus costing the company big penalties. Working from the cost perspective can mean that almost all tasks from the old schedule could be rescheduled, but the final objective (on time delivery) is attained.

Formally, we define the new optimal solution like this:

$$\mathcal{X}^*_{new} = argmax_{\mathcal{X}} \left(\sum_{r_l \in \mathcal{R}} r_l(\mathcal{X}) - \sum_{\sigma_i \in \mathcal{S}} \sigma_i(\mathcal{X}^{old} \rightarrow \mathcal{X}) \right) \quad (2)$$

where the first sum is the utility of the new solution, and the second sum is the cost one has to pay for changing the current assignments to the new ones.

For uncommitted variables, the cost is 0: they can simply choose their new optimal values, without any cost. Hard-committed variables cannot change their values anymore (one can think of it as an infi nite change cost).

Thus, what we need to optimize is the difference between the new utility and the cost associated with changing the soft-committed variables. We will show in section 6.1 how this is actually done.

6 Algorithm RSDPOP

There are two major changes from the original SDPOP. First, we add a time monitor for each agent that handles the deadlines imposed on the commitment of its variable. Second, the *UTIL* propagation is changed as far as the committed variables are concerned (either projection with cost of change for the soft-committed variables, or simply slicing for the hard-committed ones).

The *RSDPOP* algorithm is described in Algorithm 2. The following subsections explain in detail the changes we made to the *SDPOP* algorithm.

6.1 *UTIL* propagation

The *UTIL* propagation is essentially the same as in *SDPOP*, with the exception of the committed variables.

Soft Commitments and Stability Constraints Suppose X_i has already soft-committed to v_i^*. If there are some changes in the problem, and X_i needs to resend its *UTIL* message to its parent, then it can recompute it by adding the cost of change to the current $JOIN$, followed by an optimal projection along its dimension: $UTIL_{X_i}^{P(i)} = \left(JOIN_i^{P(i)} \oplus \sigma_i[v_i^*]\right) \perp_{X_i}$.

For each tuple of variables in $\{dim(JOIN_i^j) \setminus X_i\}$, all the corresponding values from $JOIN_i^j$ (one for each value of X_i) are selected. The value corresponding to $X_i = v_i^*$ is not modifi ed - no change, no cost. From all the other values corresponding to $X_i = v_i^k, k \neq *$ we substract "the cost of change": $\sigma_i(v_i^* \rightarrow v_i^k)$. We then choose the best value. Computing the *UTIL* messages like this ensures that the utility values sent by X_i are either computed by keeping the same value for X_i, or take into account the cost of change.

Hard commitments In case the variable is not eliminated immediately, it will still participate in the *UTIL* transmission. When computing its *UTIL* messages, a dead variable cannot choose its value freely anymore, so instead of an optimal projection, a slice is used: if X_i was already assigned to v_i^*, then $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)}[X_i = v_i*]$

6.2 *VALUE* propagation

Now the optimization of the local value happens only if the variable is not hard-committed. If it is soft-committed, the cost of change is taken into account. Otherwise, the variable is "fbating", and it can freely be changed to its new optimal value.

Algorithm 2: *RSDPOP - Realtime stable distributed pseudotree optimization*

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

Self-stabilizing DFS protocol: run continuously; if changes in topology, reactivate

1 after stabilization, X_i knows $P(i), PP(i), C(i), PC(i)$

Time monitor: run continuously

2 **if** *deadline reached & not yet committed* **then** commit to current best value: $X_i \leftarrow v_i^*$

3 **if** *hard commit* **then** mark as *dead*; apply policy on dead nodes

***UTIL* propagation protocol**: run continuously

4 get and store all new *UTIL* messages (X_k, $UTIL_k^i$)

5 **if** *P(i), PP(i), C(i), PC(i),* $UTIL_k^i$ *or* R_i^k *changed* **then**

6 build $JOIN_i^{P(i)} = \left(\left(\bigoplus_{c \in C(i)} UTIL_c^i\right) \oplus \left(\bigoplus_{c \in \{P(i) \cup PP(i)\}} R_i^c\right)\right)$

7 **if** X_i *is hard-committed to* v_i* **then** $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)}[X_i = v_i*]$

8 **if** X_i *is soft-committed to* v_i* **then** $UTIL_{X_i}^{P(i)} = \left(JOIN_i^{P(i)} \oplus \sigma_i[v_i^*]\right) \perp_{X_i}$

9 **if** X_i *is not committed* **then** $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)} \perp X_i$

10 send $UTIL_{X_i}^{P(i)}$ to P(i)

11 **if** X_i *is root* **then** start *VALUE* propagation

***VALUE* propagation protocol**

12 get and store in $agent\_view$ all *VALUE* messages ($X_k \leftarrow v_k^*$)

13 **if** X_i *is not committed* **then** $v_i^* \leftarrow argmax_{X_i}\left(JOIN_i^{P(i)}[agent\_view]\right)$

14 **if** X_i *is soft-committed* **then** $v_i^* \leftarrow argmax_{X_i}\left(JOIN_i^{P(i)}[agent\_view] \oplus \sigma_i[v_i^*]\right)$

15 Send $VALUE(X_i \leftarrow v_i^*)$ to all $C(i)$ and $PC(i)$

7 Experimental evaluation

Our experiments were performed on distributed meeting scheduling problems. We modeled a realistic scenario, where a set of agents working for a large organization try to jointly find the best schedule for a set of meetings. The organization itself has a hierarchical structure: a tree with departments as nodes, and a set of agents working in each department. In a realistic organization, the majority of interactions are within departments, and only a small part are across departments, and even then, normally the interactions take place in a command-chain-like fashion. We took this aspect into account while generating our test instances: with high probability we generate meetings within departments, and with a lower probability we generate meetings between agents belonging to parent-child departments.

Although this is a cooperative setting, we assume that privacy is a requirement. Therefore, among several possible ways to model this problem as a MCOP, we chose the PEAV model from [4]. Specifically, each agent A_i has a set of variables X_i^j, one for each meeting it is involved in. Each such variable X_i^j is controlled only by the agent A_i, and represents the time when meeting j of agent A_i will start (X_i^j has time slots t_k as values). There is an equality constraint connecting the equivalent variables of all agents involved in a particular meeting (all agents must agree on a start time for their meeting). If a meeting has k participants, it is sufficient to create $k - 1$ equality constraints that

connect the corresponding variables in a chain (no need to fully connect them pairwise). Since an agent cannot participate in 2 meetings at the same time, there is an all-different constraint on all variables X_i^j belonging to the same agent.

We model the utility that each agent A_i assigns to each meeting M_j at each particular time $t_k \in dom(X_i^j)$ by imposing unary constraints on the variables X_i^j; each such constraint is a vector private to A_i, and denotes how much utility A_i associates with starting meeting M_j at each time t_k. Obviously, the objective is to find a schedule s.t. the overall utility is maximized.

Efficiency of the solving process The results from Table 3 show how our method scales up with the size of the organization, and the number of meetings to be scheduled. As expected, the overall complexity is not necessarily influenced by the sheer size of the problem to be solved, but rather by its density. This shows us that this method has the potential to basically scale indefinitely, as long as the problems remain loose and with low induced width. To our knowledge, we have solved by far the largest distributed optimization problems with a complete algorithm (second to us would be [4], with 33 agents, 12 meetings, 47 variables, 123 constraints).

We also note that the number of messages recorded in Table 3 is the total number of *virtual* variable-to-variable messages. Since in this model an agent necessarily has several internal variables, this means that the number of *real* agent-to-agent messages is strictly lower than that.

Another remark is about the path the large messages travel through. We have explained in the model we have mutual exclusion constraints between all pairs of variables of each agent (an agent cannot participate to more than one meeting at a time). This actually creates small cliques in the graph, which are most likely to contribute to the high width of the problem. However, as seen in section 3.3, and shown in [6], only the high-width parts of the graph generate messages with high dimensionality. This means that actually many large messages are intra-agent *virtual* messages, thus they do not contribute to the network load.

Agents	Meetings	Variables	Constraints	Width	Messages	Max. message size	Time to sol. (s)
30	14	44	52	3	95	512	2.1
40	15	50	60	4	109	4096	7.2
70	34	112	156	5	267	32768	21.6
100	50	160	214	6	373	262144	43.4
200	101	270	341	6	610	262144	72.3

Table 3. RSDPOP tests on meeting scheduling.

Solution stability To model a realistic, dynamic scenario, we consider again the meeting scheduling example. This time however, we assume that the agents must also travel to the meetings they will attend. Thus, they must arrange for transportation (airplane tickets) and accommodation (hotel reservations). There is obviously a cost for canceling these arrangements, as some tickets may be non-refundable, some hotels may charge the room price for one night in case of no-show, etc. This is modeled by a stability constraint on each variable whose reassignment involves a cost. We simulate a dynamic

problem by changing the unary constraints of the agents, meaning their preferences evolve dynamically.

The task is to find the best possible schedule at each time, taking into account the new preferences of the agents, but also the cancellation costs that are incurred by changing the starting times of the meetings that the agents have committed to.

We have experimented on instances of problems with 40 agents, 76 variables and a total of 256 constraints. We have simulated a dynamic evolution of the problem by gradually introducing changes in successive instances of the problem. The changes amounted to an additional 10% of the variables in each execution. These variables were considered soft-committed, s.t. in the next execution of the algorithm the cost penalties of changing their values had to be taken into account.

We have tested two versions of the algorithm: an *SDPOP*-like one which always optimizes the current state, while trying to minimize assignment changes, and the *RSDPOP* one which optimizes according to Equation 2. The results are in Table 4; each cell presents the results in the form *RSDPOP*/*SDPOP*. We define the distance between two solutions as the number of different assignments. As expected, as the change rate increases, so do the costs of change, and the solution distances. We notice that RSDPOP makes more adjustments than SDPOP to the solutions it previously generated. However, by taking the costs explicitly into account, RSDPOP incurs less penalties, and obtains better overall solutions.

% vars changed	Optimal sol.	Cost penalties	Solution distance
0 / 0	1219 / 1219	0 / 0	0 / 0
10 / 10	1217 / 1207	-3 / -38	6 / 2
20 / 20	1217 / 1208	-3 / -14	10 / 2
30 / 30	1212 / 1198	-9 / -83	24 / 3
40 / 40	1201 / 1173	-13 / -80	27 / 7
50 / 50	1202 / 1180	-19 / -95	31 / 9

Table 4. Tests on dynamic meeting scheduling: RSDPOP vs SDPOP

8 Practical issues

Essentially, the costs associated with the stability constraints come from a separate, business-specific process. Possible examples include extra fuel and loading/unloading costs for a vehicle routing problem, contractual penalties, etc. Thus, we think it is a great advantage that it is possible for these costs to be specified as real numbers, as opposed to setting weights to the variables that we want kept to their current values.

Commitment deadlines are straightforward, and can come directly from contracts between partners, or from internal processes which need to be completed on time.

8.1 Garbage collection

When purging variable X_i, one needs to maintain the effect the hard-committed variable has on its neighbors. This is achieved by replacing each binary relation X_i has with its neighbors, with a corresponding unary constraint placed on the respective neighbor. Example (figure 1): if X_8 has been hard-committed to v_8^2, then X_8, R_8^3 and R_8^1 can be

deleted, and the unary relations $R_3(8) = R_8^3[v_8^2]$ and $R_1(8) = R_8^1[v_8^2]$ are placed on X_3 and X_1 respectively. This completely summarizes X_8's effect on the problem, and does not change anything w.r.t. the optimal solution.

Removing a variable could trigger the DFS protocol to adjust the current *DFS* tree. This can be a performance hit, so a removal policy must be tailored for the problem at hand. Possible policies are e.g. a timeout mechanism (variables hard-committed for a long time are purged), or a threshold mechanism (when the number of hard-committed variables in the problem exceeds a threshold).

9 Conclusions and future work

We define the *distributed, continuous-time combinatorial optimization problem*. We propose a general, cost-based metric for *solution stability* in such systems. We present the first mechanism for combinatorial optimization that guarantees optimal solutions in dynamic environments, with respect to this metric. In contrast to current approaches, our mechanism is a lot more flexible and allows for a much finer-grained vision of time: each variable of interest can be assigned *its own commitment deadlines*, allowing for a *continuous-time* optimization process. The experimental results show that this approach gives good results for low width, practically sized dynamical optimization problems.

As future work we envisage addressing issues like robustness, different garbage collection policies, and analyzing ways of dealing with very tight deadlines.

References

1. Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.
2. A. Dechter and R. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-88*, pages 37–42, St. Paul, MN, 1988.
3. Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
4. 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-04*, 2004.
5. Adrian Petcu and Boi Faltings. Approximations in distributed optimization. Technical Report 2005018, EPFL, Lausanne,Switzerland, May 2005.
6. Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, Edinburgh, Scotland, Aug 2005.
7. Adrian Petcu and Boi Faltings. Superstabilizing, fault-containing multiagent combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, Pittsburgh, Pennsylvania, USA, July 2005.
8. G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-94*, pages 307–312, Seattle, WA, 1994.
9. R.J. Wallace and Eugene C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Proceedings of CP98*, 1998.