

Implementing explained global constraints

Étienne Gaudin¹, Narendra Jussien², and Guillaume Rochart^{2,3}

¹ Bouygues e-lab

1 av. Eugène Freyssinet – F-78061 St Quentin en Yvelines Cedex

² École des Mines de Nantes, Département Informatique

4, rue Alfred Kastler - B.P. 20722 – F-44307 Nantes Cedex 3

³ LINA, FRE CNRS 2729

2, rue de la Houssinière – B.P. 92208 – F-44322 Nantes Cedex 3

egaudin@bouygues.com, {jussien, grochart}@emn.fr

1 Introduction

Constraint satisfaction problems (CSP) [22] have proven to be an efficient model for solving many combinatorial and complex problems. Moreover, recurring patterns and sub-problems in those problems are now handled through global constraints [2, 4, 18]. Global constraints have been a good advocate for using constraint programming techniques to solve real-life problems.

Explanations (specializing (A)TMS [7]) and generalizing nogoods [21]) have been initially introduced to improve backtracking-based algorithms [8]. However, they have been recently used for many other purposes [11] including new solving techniques [13, 14], dynamic constraint solving [23, 6] and user interaction [15]. Explanations represent an explicit and limited *trace* of the behavior of the solver. They are mainly used both for dynamic solving and user-interaction purposes. Maintaining and computing explanations may be particularly costly. Fortunately, explanation-based algorithms efficiently solve problems thanks to mechanism reducing *thrashing* during the resolution.

Introducing both explanations and global constraints in a constraint solver is a real challenge. Indeed, three points may be kept in mind: (1) providing precise and meaningful explanations to be offered to the user extracting the locality of part of the reasoning of the global constraint, (2) altering the efficiency of the original constraint as few as possible *i.e.* developing algorithms within the constraints efficient for both filtering and explanation capabilities and (3) providing algorithmic tools for both incrementality (taking into account new events from filtering) *and* decrementality (in order to be able to replace backtracking with repair-based techniques as explanation-based solvers usually offer).

In this paper, we illustrate those three points with a global constraint encapsulating flow theory that maintains a feasible flow in a given network, similar to the flow constraint in [5]. Such a constraint is useful for several real-life problems involving resource allocation and is well suited for illustrating the challenges of embedding explanations. In the following, we first set the context of our study in Section 2 and introduce the flow constraint (Section 3). Then, the three points are successively addressed in Sections 4, 5 and 6 keeping the flow constraint as an illustration. Finally, some experiments are introduced showing the interest and

the efficiency of our approach in Section 7 with respect to the three challenges defined below.

2 Context

2.1 Constraint Satisfaction Problems

Following [22], a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function $\text{var} : C \rightarrow \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V . Constraints are defined by their set of allowed tuples providing the semantic part of the constraint network. A tuple is a set of couples (var, val) where $\text{var} \in \text{var}(c)$ and $\text{val} \in d_{\text{var}}$ its domain (set of allowed values for var). Notice that domains are considered here as unary constraints.

2.2 Global constraints

In practice, constraints are seldom provided as a set of allowed tuples. Usually, arithmetic or symbolic expressions are used to describe the constraints of a given CSP. Moreover, *global constraints* [2, 18] are now often used to encompass several recurring patterns arising in optimization problems. Following [4] we will focus here on operationally global constraints: *Operational globality considers both a constraint and a consistency notion. The constraint is said to be global if there exists no decomposition scheme for which the consistency notion removes as many local inconsistencies as on the original constraint. This concept is important because it compares the pruning of the constraint and its decompositions wrt a consistency notion. If the constraint prunes more than its decompositions, then, from a CSP standpoint, it is operationally global, since the closure of the decomposition cannot recover the consistency achieved on the original constraint.*

Such a constraint, from an implementation point of view, usually involves specific filtering algorithms that maintain a given support structure⁴ in order to provide useful information for powerful domain reductions or early identification of failures. For example, for the `allDifferent` constraint [17], a reference matching is usually maintained which is used to compute strongly connected components for the sake of the filtering algorithms.

2.3 Explanations for constraint programming

An explanation [11] for constraint programming contains enough information to justify a decision (throwing a contradiction, reducing a domain, etc.): it is composed of the constraints and the choices made during the search which are sufficient to justify such an inference.

⁴ Notice that the support structure may not be explicitly maintained within the constraint but algorithmic tools are usually provided in order to be able to access it in a reasonable time.

Definition 1. An *explanation* of an inference (\mathcal{X}) consists of a subset of original constraints ($\mathcal{C}' \subset \mathcal{C}$) and a set of instantiation constraints (choices made during the search: d_1, d_2, \dots, d_k) such that: $\mathcal{C}' \wedge d_1 \wedge \dots \wedge d_n \Rightarrow \mathcal{X}$.

An explanation e_1 is said to be more precise than explanation e_2 if and only if $e_1 \subset e_2$.

The more precise an explanation, the more useful it is. However, this is not a sufficient condition. There can be numerous minimal explanations. This is why we have chosen to find a compromise between precision and ease of computation.

Computing explanations The most interesting explanations are those which are minimal regarding inclusion. Those explanations allow highly focused information about dependencies between constraints and variables. Unfortunately, computing such an explanation can be exponentially costly. A good compromise between precision and ease of computation consists in using the solver embedded knowledge to provide explanations [12]. Indeed, constraint solvers always know, although it is scarcely explicit, *why* they remove values from the domain of the variables. By making that knowledge explicit, quite precise and interesting explanations can be computed as constraint solvers are supposed to efficiently perform their task! Therefore, explanations strongly depend both on the constraint solver at hand and on the way the problem is modelled.

2.4 User accessible explanations

The primary usage of explanations is to provide the user some feed-back about resolution. Precise and meaningful explanations are therefore required. Moreover, some kind of user interface for explanations is also required. Indeed, explanations as we introduced them, reflecting the behavior of the solver, may be far from the user representation of the problem. Tools are needed to translate that internal information to a problem related representation [15].

2.5 Explanations for dynamic constraint solving

Solving dynamic constraints problem has led to different approaches [23]. Two main classes of methods can be distinguished: proactive and reactive methods. On the one hand, proactive methods propose to build robust solutions that remain solutions even if changes occur. On the other hand, reactive methods try to reuse as much as possible previous reasonings and solutions found in the past. They avoid restarting from scratch and can be seen as a form of learning. Using explanations falls in such a reactive techniques [11, 6].

Explanations for dynamic constraint retraction Dynamic constraint retraction is basically a two-step process [6]: enlarging the current environment (in order to *undo* past effects of the retracted constraint which are the events whose explanation contains the retracted constraints) and restoring a given consistency for the resulting constraint network.

Search as a dynamic process Using explanations to help solving (as in *Dynamic Backtracking* [8] or in `mac-dbt` [13]) modifies the way search is done. In those algorithms (as in `decision-repair` [14]) backtracking is replaced by a repair mechanism as handling contradiction is no longer handled by getting back to an already explored safe situation. Contradiction can now be precisely explained allowing modifications in the set of decisions (usually variable assignments) performed in a surgical way.

Those recent algorithms introduce new events in a constraint solver (domain enlargement) and a new kind of incrementality: decremental requirements.

2.6 Global constraints and explanations: operational requirements

Implementing global constraints in an explanation-aware setting leads to three challenges:

- *computing (quasi-)minimal and user meaningful explanations.* Explanations computing for and by a global constraint should provide as much information as possible *ie*, as paradoxal as it may appear, explanations should be as local as possible. Moreover, they should be easily interpreted in terms of the underlying theory of the constraint in order to be translated into comprehensible terms for a user. Consider for example the `allDifferent` constraint where a value is removed from the domain of a variable when the variable and its value are in two different strongly connected components in the graph representing the current reference matching. In order to provide an explanation for that situation some more information need to be computed and a theoretical analysis should be performed [20]. Usually, the support structure need to be enhanced in order to provide efficient tools for efficient explanations during propagation⁵.
- *efficiency of the constraint must be preserved.* Adding explanations capabilities into a global constraint must not degrade its performance to an unbearable level. The key point for such constraints is to *incrementally* maintain the enhanced support structure. Recomputing from scratch the information upon each value removal should be avoided in order to maintain a low computational cost of the filtering algorithms. Indeed, the idea is to be able to design a propagation algorithm efficient both for filtering and explanation computation.
- *incrementality and decrementality should be provided.* As with explanations, backtracking can easily be replaced with some kind of repair mechanisms, constraints should be able to provide both incremental AND decremental algorithms need to be designed in order to provide efficient and powerful constraints for new search algorithms.

In the following, we will illustrate those three points with a global constraint that maintains a feasible flow in a given network.

⁵ Notice that this is not always the case: see the `stretch` constraint for example[20].

3 Application to the flow Constraint

We chose here to illustrate the way global constraints should be instrumented for explanation-based solvers with a flow constraint. Such a constraint is meant to maintain a feasible flow in a given network. We recall some information about flow theory and introduce the global constraint.

3.1 Flow Theory

The subsection content is based on the book “Network Flows” by Ahuja, Magnati and Orlin [1]. We only give a quick survey of the main definitions and properties used in this paper. The interested reader will find much more details and all proofs in it.

Let's note $G = (N, A)$ an acyclic directed graph with N the set of nodes and A the set of arcs.

Definition 2 (Feasible Flow). *The following equations model a feasible flow problem on G between two specific nodes of N , the source s and the sink t :*

$$\sum_{j:(i,j) \in A} x_{ij} - \sum_{j:(j,i) \in A} x_{ij} = \begin{cases} v & \text{for } i = s \\ 0 & \forall i \in N \setminus \{s, t\} \\ -v & \text{for } i = t \end{cases} \quad (1)$$

$$\forall (i, j) \in A, l_{ij} \leq x_{ij} \leq u_{ij}, l_{ij} \geq 0 \quad (2)$$

Equations (1) ensure flow conservation at each node, (2) impose that the amount of flow going through each arc is between its lower bound l_{ij} and upper bound u_{ij} . We refer to $x = \{x_{ij}\}$ satisfying (1) and (2) as a feasible flow and the value of v as the value of the flow. Figure 1 gives an example of a feasible flow problem.

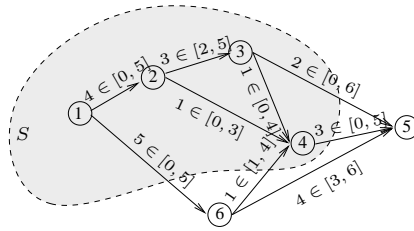


Fig. 1. Flow example in an acyclic oriented graph

Let's now introduce cuts and their capacity. A cut is a partition of the set of nodes N into two subsets. Flow theory is interested by a specific type of cut, the $s - t$ cut.

Definition 3 ($s - t$ Cut). *An $s - t$ cut (S, \bar{S}) is a partition of the set of nodes N into two subsets S and $\bar{S} = N - S$ such as $s \in S$ and $t \in \bar{S}$.*

A cut is denoted by (S, \bar{S}) , we use also this notation to refer to the set of arcs (i, j) such as $i \in S$ and $j \in \bar{S}$. The capacity of a cut gives bounds on the amount of flow we can send from one set of the nodes to the other one. It plays a central role for explanation in a flow constraint (see Section 4).

Definition 4 (Capacity of an $s - t$ Cut).

$$C(S, \bar{S}) = \sum_{(i,j) \in (S, \bar{S})} u_{ij} - \sum_{(i,j) \in (\bar{S}, S)} l_{ij} \quad (3)$$

For example, in Figure 1, the capacity of the cut $(\{1, 2, 3, 4\}, \{5, 6\})$ is 15. It is equal to the difference between the sum of the maximum capacities $u_{16} = 5$, $u_{45} = 5$ and $u_{35} = 6$ and the minimum capacity $l_{14} = 1$.

Let us introduce two properties linking cut capacity and flow value v .

Property 1 *The value v of any flow x is less than or equal to the capacity of any $s - t$ cut in the network*

This property is quite intuitive as any flow should pass through every $s - t$ cuts and therefore can not exceed their capacity.

Property 2 (Max-Flow Min-Cut) *the maximum value v of the flow x from s to t equals the minimum capacity among all $s - t$ cuts.*

This last property shows the dual aspects of flow problems, any maximum flow problem is equivalent to a minimum cut problem.

3.2 Semantics

The flow constraint studied in this paper is a direct translation in a global constraint of the feasible flow problem (see definition 2). Compared to Chip flow constraint [5], it is restricted to flow feasibility on acyclic graph. The following call `flow(pb, G, s, t, V)` in a CHOCO program [16] defines a flow on G between s and t of value v where:

- G is a list formulation of the graph G ; an arc (i, j) is defined by couple (j, X_{ij}) in the list $G[i]$ where X_{ij} is the variable denoting the flow going through the arc $(i, j) \in A$ with its domain $d_{X_{ij}} = [l_{ij}..u_{ij}]$,
- s and t are the index of the source and sink nodes s and t in the list G ,
- V is the variable denoting the flow value v between source and sink,

The feasible flow given in Figure 1 corresponds to the following CHOCO call.

```

Problem pb = new Problem();
IntVar X12 = pb.makeEnumIntVar("X12", 0, 5);
...
CapaEdge[] G = new CapaEdge[][]{
    {new CapaEdge(X12,2), new CapaEdge(X16,6)}, //(1,i) arcs
    {new CapaEdge(X23,3), new CapaEdge(X24,4)}, //(2,i) arcs
    ...
};
pb.post(flow(G, 1, 5, V));

```

The following section presents propagation principles and flow algorithms used in the flow constraint.

3.3 Flow Algorithms

To implement a Flow constraint, propagation algorithms should answer the following questions:

1. Is the support set empty or not?
2. What are the bounds of V according to G ?
3. What are the bounds of X_{ij} of G according to V or other X_{ij} variables?

The first point is equivalent to solving the Feasible Flow Problem of (2). The two last points are related to Minimum and Maximum Flow Problems with Minimum Capacity.

Feasible Flow Problem Finding a feasible flow is obvious when there is no minimum capacity on arcs (*i.e.* $\forall(i, j) \in A, l_{ij} = 0$). For such a graph an empty flow is a feasible one. When some l_{ij} are greater than zero, Berge gives in [3] an algorithm based on solving a maximum flow problem on a transformed graph. This graph has two interesting property:

1. a feasible flow x exists in G if and only if the value of a maximal flow in the transformed graph saturates arcs of its source and sink;
2. all arc's minimum capacity in the transformed graph is zero so standard maximum flow algorithm could be applied.

This method is used straightforwardly to find a feasible flow or to prove that the support of a flow constraint is empty. In the rest of the paper, we use Φ to denote a support, *i.e.* a feasible flow of G .

Minimum and Maximum Flow Problem with Minimum Capacity The existence of minimum capacity on arcs makes computing minimum or maximum flow more complex. A two stage algorithm should be applied:

1. find a feasible flow Φ using the previous algorithm;
2. applying a maximum flow algorithm on the residual network⁶ containing Φ .

For finding a maximum flow, one just has to solve a maximum flow problem on the residual network containing Φ . Finding minimum flow is also quite simple. One has to solve a maximum flow problem on the same residual network but from the sink t to the source s . Intuitively, this last problem removes as much flow as possible between t and s while maintaining a feasible flow. Finally, the resulting flow is feasible and minimum.

⁶ The residual network is a common graph representation that contains both the flow and the remaining flow of each arcs. It is used in most graph algorithms to maintain incrementally a flow.

Propagation principles Two types of propagation have been defined in the flow constraint:

1. **Flow conservation propagations** *inside* $\{X_{ij}, \forall (i, j) \in A\}$ *and between* $\{X_{ij}, \forall (i, j) \in A\}$ *and* V
 These propagations are local to each node. They insure the flow conservation constraint (1) by applying the standard (in CP solver) bound propagation rules of linear formula [9].
2. **Global flow propagations** *from* $\{X_{ij}, \forall (i, j) \in A\}$ *to* V
 They are global to the graph and based on algorithms described below. They check the existence of a feasible flow Φ (*i.e.* a support) and maintain the value of the flow in the graph, *i.e.* the domain of the V variable, by solving a minimum and a maximal flow problems⁷.

A small modification of the original graph G is used to improve propagations without any change in the algorithms. We add a new node, denoted st for super sink, with one arc from sink t to this super sink st with variable V as the arc domain capacities. This transformation imposes that any feasible flow Φ should be in the domain of V and so insures bound consistency of the V variable.

The propagation algorithms are combined in the flow constraint according to the complexity of the underlying algorithms. Here is a sketch of the algorithms. Propagations based on standard linear equation propagations are called first. After reaching a fix-point without contradiction, the global flow propagations is called. These last propagations compute new bounds of the flow variable V . If they are different from current ones, two situations could occur: if the computed bounds are inconsistent with current bounds of the flow variable V , a contradiction should be triggered; if the computed bounds reduce the domain of variable V , the new bounds should be propagated.

We detail in Section 5 how this process could be speed up using a incremental version of the global flow propagations.

4 Accurate explanations for the flow constraint

Explaining efficiently global constraints is a challenging task: generated explanations must be as precise as possible without slowing down the resolution. This section presents how to generate precise explanations for the flow constraint thanks to nearly cost less algorithms based on the cut notion.

Since the flow constraint propagates on variables bounds, we will note $expl(x \geq v)$ (resp. $expl(x \leq v)$) the explanation of the lower bound of variable x and more precisely the reason why the variable x must be at least equal to v (resp. the reason why x must be less than or equal to v). With respect to the flow constraint, two kinds of filtering removals are deduced: the flow conservation rules and the minimum, maximum or feasible flow properties.

⁷ We choose not to solve a minimum and a maximum flow problems on each arc to maintain X_{ij} bounds as it seems to be too costly. These bounds are only update by the flow conservation propagations.

4.1 Flow conservation explanations

As illustrated on equation (1), flow conservation rules are based on the Kirchoff law. Since flow in the network is considered as positive values on the arcs, the following rule can be used : $\forall i, k \in N \setminus \{s, t\}, (i, k) \in A, u_{ik} = \sum_{j:(j,i) \in A} u_{ji} - \sum_{j \neq k:(i,j) \in A} l_{ij}$.

This rule makes the explanation explicit: the lower and upper bounds involved in this equation imply the value of the new upper bound of x_{ik} . The explanation of this bound is then $(\forall i, k \in N \setminus \{s, t\}, (i, k) \in A)$:

$$\text{expl}(\mathbf{x}_{ik} \leq u_{ik}) = \bigcup_{j:(j,i) \in A} \text{expl}(\mathbf{x}_{ji} \leq u_{ji}) \cup \bigcup_{j \neq k:(i,j) \in A} \text{expl}(\mathbf{x}_{ij} \geq l_{ij}) \quad (4)$$

This result can easily be generalised to all filtering rules based on this law.

4.2 Maximal flow explanation

As we are dealing with global properties, computing precise explanations is not easy. We could actually use the trivial explanation (union of all variables explanations in the network) but this is far from being precise.

In order to keep things clear, all flows are assumed to be feasible in this section. Properties 1 and 2 guarantee that the maximum value of the flow from a source s to a sink t equals the minimum cut capacity. Computing such a cut is nearly costless as soon as a maximal flow is found. Thus a minimum capacity cut can be used to explain the maximal flow in a network. The capacity of a cut is defined on Equation 3: $C(S, \bar{S}) = \sum_{(i,j) \in (S, \bar{S})} u_{ij} - \sum_{(i,j) \in (\bar{S}, S)} l_{ij}$.

The equation shows that the only way to increase the upper bound of the flow across the network is to modify one of these bounds. The following property is now immediate :

Proposition 1 *Given a network and a minimal $s-t$ cut (S, \bar{S}) in this network, an explanation for the maximum flow across the network can be defined as follows (where $\text{expl}(\mathbf{x}_{ij} \leq u_{ij})$ and $\text{expl}(\mathbf{x}_{ij} \geq l_{ij})$ are maintained bound explanations):*

$$\text{expl}(\mathbf{v} \leq C(S, \bar{S})) = \bigcup_{(i,j) \in (S, \bar{S})} \text{expl}(\mathbf{x}_{ij} \leq u_{ij}) \cup \bigcup_{(i,j) \in (\bar{S}, S)} \text{expl}(\mathbf{x}_{ij} \geq l_{ij}) \quad (5)$$

For instance, in the network illustrated on Figure 1, when a maximal flow is reached, the cut $S = \{1\}$ is computed. This means that the explanation should contain the upper bound explanations for $x_{1,6}$ and $x_{1,2}$. Such an explanation is sufficient for justifying the new upper bound of the flow across the network from 1 to 5 - here this upper bound equals $u_{1,5} + u_{1,6} = 10$. The computation of such an explanation is quite easy when a maximal flow is known: the algorithm compute reachable nodes from source in the residual graph, and the cut contains all arcs between these nodes and nodes outside ($O(m \cdot n)$ complexity). This algorithm does not guarantee minimal explanations, but it provides an algorithm with an acceptable computing overhead. Since explanations can be exploited to dynamically remove constraints, they need to be always available for each variable domain. This makes the use of algorithms like QUICKXPLAIN [10] (based on dichotomic search of minimal conflict set over constraints) prohibitive.

The same principles are used to compute explanations for minimum flow.

4.3 Feasible flow

When there is no feasible flow in the graph, a contradiction is raised. This situation needs to get an explanation. As we saw earlier, searching for a feasible flow is the same as searching for a maximum flow in a modified graph. Therefore, the same idea applies. The only issue is to take into account the modifications the graph in the explanations (see [19] for details).

5 Efficient filtering algorithm and explanation algorithm

The main difficulty of the implementation of a global constraint lies in the dynamic nature of CP solvers. Constraints must react to solver events such as modifications of the domain of their variables, in order to check their feasibility and deduce some domain reductions or inconsistency. A keen constraint implementation must trigger as less propagation rules as possible for any set of events. A solution is to filter solver events with the constraint support structure and to use incremental algorithms.

5.1 Filtering events with flow constraint support structure

The `flow` constraint reacts only to bound modification events of its variables: an increase/decrease of the lower/upper bound of an arc capacity (variables $\{X_{ij}\}$) and an increase/decrease of the minimum/maximum flow (variable V). For both events, we use the same two stage mechanism[16]: flow conservation propagations rule is triggered immediately and global flow propagations are delayed.

For such delayed propagations, triggered events are filtered through constraint support structure. For the `flow` constraint, we choose a residual network as the support structure. At any fix-point, it contains a feasible flow Φ . Two reactions can happen according to Φ :

1. The new domain bounds are consistent with current support flow Φ , *i.e.* x_{ij} the flow on arc (i, j) in Φ is still in the reduced domain $d_{x_{ij}}$. Triggering a propagation rule is useless, as Φ is still a support.
2. It is incompatible with current flow Φ , post a new event to the solver in order to awake the constraint for finding new feasible flow and, if necessary, computing the new minimal/maximum flow.

5.2 Finding incrementally a new feasible flow

The second item is done incrementally. The basic principle is to start from current flow support Φ instead of starting with empty flows. We suppose that making the few changes to obtain a new valid support is less costly than starting from scratch.

As explained in Section 3.3, the graph transformation of Berge [3] was designed to search a feasible flow from a null flow by adding new arcs in order to fill a lack of flow in some nodes and to remove an excess of flows in some other

ones. It can be generalised to add or remove some flows on any arcs from any initial flow and so, used to change any flow into a new one consistent with new bounds on arc capacities.

To build the transformed graph, one should start from any feasible flow and:

- initialise $\forall i \in N, b(i) = 0$ a b vector representing lack or excess of flow at each node;
- in order to add q units of flow on arc (i, j) (because the lower bound is more than the support value), remove q from $b(i)$, add q to $b(j)$ and update the structure;
- in order to remove q units of flow on arc (i, j) (because the upper bound is less than the support value), add q to $b(i)$, remove q from $b(j)$ and update the structure.

When all the updated flows have been taken into account, each node i with $b(i)$ positive (resp. negative) is linked to a new source node (resp. sink node) with a maximum capacity on the arc equal to $b(i)$.

A feasible flow compatible with all the added and removed flows exists if and only if maximum flow saturates source and sink arcs. The proof of this property is a direct extension of the one used in [1] for the feasible flow.

To use this algorithm for an incremental version of propagation rules, one has to add in the support structure a b vector. For all events on the bounds of $X_{i,j}$ and V variables, the flow constraint should update the support graph and, if this new bound is inconsistent with current flow Φ , the b vector.

Each filtering decision must be precisely explained. As Section 4 shows, computing explanations depends on cut. Fortunately this is nearly costless as soon as a maximum flow (in G or G') is available thanks to the Φ support.

6 Decremental data structures

Incremental algorithms presented in last section depends on up-to-date data structures. CP solvers provide backtracking mechanisms that maintain past states of the structure. In dynamic solving (like explanation based algorithms) decremental process must be included into the constraint to ensure the structure to be always consistent with the current state of variables.

6.1 Classical solving

All information depending on the state of variables used by the filtering algorithm must be up-to-date. This compels the following data to be maintained whenever the state is modified:

- the support Φ (a feasible flow), since all filtering algorithms depend on them;
- all data depending on the state of variable, like b values.

With classical backtrack, the state reached after a contradiction is always a past state. This means that the data structure only needs to be stored before trying new search decisions: classical trailing method is sufficient.

6.2 Dynamic solving

In a dynamic context (where a constraint or a decision can be undone even if this is not the last posted one), trailing is useless as the state reached after a contradiction is not necessarily a past state anymore. That means that the constraint must be able to update itself its data structure upon repairing.

With the `flow` constraint, removing constraints does not make the current data structure false, since a feasible flow in a network is still feasible in a less constrained network. This means that in many cases, the constraint only has to update maximum flow and minimum flow from past supports.

However, in 5.1 we saw that the support is not always up-to-date since compatibility check of the support is delayed for the sake of efficiency. Thus it is not possible to use the current support to build a new one. The following algorithm scheme is used to overcome this issue:

- *when an event is triggered, the event is stored in a stack;*
- *when a contradiction occurs, the stored support is restored and updated according to stored events;*
- *when a feasible flow is found, the support is stored and the stack cleaned.*

Similar methods may be useful for all constraints involving delayed filtering rules. It guarantees that the obtained structure is consistent with the new state by updating support with past events, while still using incremental filtering algorithms.

7 Experiments

7.1 Problem presentation

Main constraints The `flow` constraint is particularly handy to model resource affectation problems. This is why an employee scheduling problem – inspired from a Bouygues staff management problem – is presented here. In this employee scheduling problem, one wants to schedule team of employees in a weekly planning. Two concerns should be affected:

- a *day off*: each team has a day off between Monday and Saturday;
- a *shift*: each team works according to predefined shifts.

The problem must checks the company loads are respected – enough teams should work according to the load of work – *and* the assignments should be fair – a team cannot be affected to the same shift or day off too many times.

For example, the bounds illustrated on Table 1 are used for the bench in the next section. In this instance, a team cannot be affected more than 3 times to the same day off, 2 teams should be affected to the evening shift each week, etc.

Additional constraints To force equitable affectation, some other sequencing constraints are added. For instance, in this bench, a team cannot be affected to the evening shift during two consecutive weeks. In real life problems, other constraints such as each team is affected to the Saturday day off at least every five weeks, must be added.

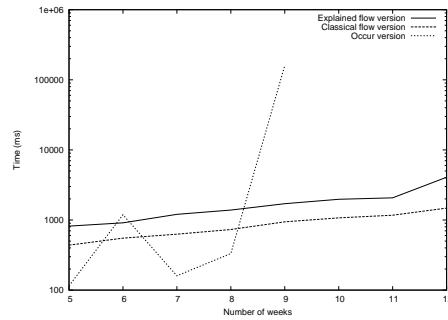
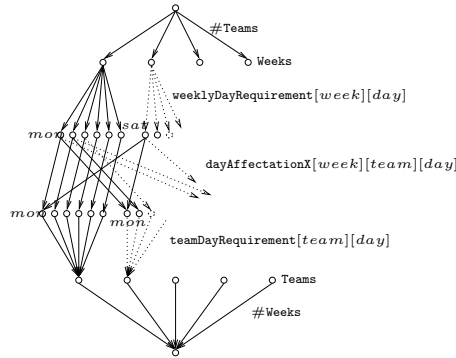
Table 1. A simple instance of an employee scheduling problem

	Day affectation						Shift affectation				
	<i>mon</i>	<i>tue</i>	<i>wed</i>	<i>thu</i>	<i>fri</i>	<i>sat</i>	<i>early</i>	<i>morning</i>	<i>day</i>	<i>afternoon</i>	<i>evening</i>
Team min.	0	0	0	0	0	0	0	0	0	0	0
Team max.	3	3	3	3	3	3	4	4	4	4	5
Week loads	1	1	1	1	1	1	1	1	1	1	2

7.2 Problem modelling

This problem can be modelled with numerous basic **occurrence** constraints. However the problem of affecting both days off and shifts can be considered as two independent **flow** constraints. Indeed let **Teams** be the set of teams of employees, **Weeks** the set of all weeks in the problem, **dayAffectationX** variables stating if a day is affected given a week and a team, and **weeklyDayRequirement** and **teamDayRequirement** affectation bounds for each day.

Then the problem of finding a day affectation for each team and week is equivalent of finding a flow in the network illustrated on Figure 2(a). Indeed, the first level of nodes constrains that each week $\#Teams$ days off must be affected (this means that each team must have a day off per week). The second level constrains the number of days off that can be affected depending on the day in the week (for instance if the company has a lot of work on Monday, no day off should be affected on Monday). The third level is composed of affectation variables modelled as presented before. And the fourth and fifth levels are the symmetric levels for teams requirement (a team can have specific bounds for day affectation: for instance, a team should be affected at least three times to the Saturday day off). Shift affectations are modelled with similar constraint.



(a) A flow model

(b) Experimentation results

Fig. 2. An employee scheduling problem

7.3 Bench results

This bench has been implemented in several versions and tested on at least ten runs: an **occurrence** model without explanations using a classical arc-consistency and backtracking algorithm in Java version of Choco [16], a **flow**

model without explanations using the same algorithms and a `flow` model with explanations using the `mac-dbt` algorithm [13] and `Palm`[12].

The results on Figure 2(b) show that the `flow` model is much more stable than the `occurrence` one. Indeed, as long as the problem is simple enough, the poor propagation of the `occurrence` constraint model is sufficient to find a solution. Furthermore, since propagation is really quick, found results are quite good compared to the `flow` model. But as soon as the problem becomes difficult enough (due to some conflicts between different `occurrence` constraints modelling subparts of the problem), this model is not efficient anymore: it needs a longer time to repair past decisions when a contradiction occurs.

Results show that explained version of the model is almost as efficient as the classical `model` whereas explanations need to be computed and maintained. Such efficiency is mainly due to precise explanations avoiding thrashing during searching a solution *and* a decrementality algorithm avoiding to compute a new support from scratch after each contradiction. Some further experimentations will be led so as to check if such a property is still true with complex search algorithms (with custom branchings for instance).

7.4 Explanations and user interaction

This application is supposed to be used by a final user. It means that when no solution is found, the user needs as much information as possible. Using explanations with the `flow` constraints provides two kinds of information: First, an explanation provides explicit inconsistent set of constraints: it is a precious information for the final user to modify the requests or to localise where the problem comes from, even if explanations must be modified to make them user-friendly [15]. Then, since the `flow` constraint is a global one involving numerous variables, the user may be interesting by a inconsistent subpart of the network; if this constraint is the only one responsible for a contradiction, instead of giving an explanation as introduced in definition 1, it can directly indicates variable bounds implying such a contradiction so as to point out bounds to modify.

8 Conclusion

In this paper, we introduced important points to be addressed when considering introducing explanations within global constraints. We illustrated our proposal with a `flow` constraint and provided experimental evaluation of our approach. To generate meaningful explanations, a theoretical study must be led to provide explanations as precise as possible, while ensuring a certain efficiency of the algorithm for practical purposes.

Our approach is particularly well designed for real complex problems as the bench showed. Further experimentations will be led to test this approach on big instances with real search algorithms.

Last, we already successfully instrumentated other global constraints (namely `stretch`, `alldifferent`, `gcc`) following the guidelines presented in this paper. More generic patterns could be covered: cyclic graphs or networks with associated cost to each edge. Such an explained global constraint menagerie may be useful for many optimisation problems.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, New York, 1993.
2. Nicolas Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
3. Claude Berge. *Graphes*. Gauthier-Villars, troisième édition, 1983.
4. Christian Bessière and Pascal Van Hentenryck. To be or not to be... a global constraint. In *CP'03*, 2003.
5. Alexander Bockmayr, Nicolai Pisaruk, and Abderrahmane Aggoun. Network flow problems in constraint programming. In *CP'01*, pages 196–210, 2001.
6. Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03*. AAAI press, 2003.
7. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
8. Matthew Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
9. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
10. Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.
11. Narendra Jussien. The versatility of using explanations within constraint programming. Research Report 03/4/INFO, École des Mines de Nantes, France, 2003.
12. Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, pages 118–133, Singapore, 2000.
13. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'00*, pages 249–261, 2000.
14. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139, July 2002.
15. Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.
16. François Laburthe. Choco: implementing a cp kernel. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, Singapore, 2000.
17. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelfth National Conference on AI*, pages 362–367, Seattle, 1994.
18. Jean-Charles Régin. Global constraints. Fourth international workshop CA-AI-OR, 2002.
19. Guillaume Rochart and Narendra Jussien. Explanations for a flow constraint (in French). Technical report, École des Mines de Nantes and Bouygues SA, 2003.
20. Guillaume Rochart, Narendra Jussien, and François Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, 2003.
21. Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problem. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
22. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
23. Gérard Verfaillie and Narendra Jussien. Dynamic constraint solving. In *CP'03*, 2003. Tutorial - online notes available <http://www.emn.fr/jussien/CP03tutorial>.