An Efficient Generic Network Flow Constraint

Robin Steiger
LIA — École Polytechnique
Fédérale de Lausanne
Lausanne, Switzerland
robin.steiger@epfl.ch

Willem-Jan van Hoeve
Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA
vanhoeve@
andrew.cmu.edu

Radosław Szymanek
LIA — École Polytechnique
Fédérale de Lausanne
Lausanne, Switzerland
radoslaw.szymanek@
gmail.com

ABSTRACT

We propose a generic global constraint that can be applied to model a wide range of network flow problems using constraint programming. In our approach, all key aspects of a network flow can be represented by finite domain variables, making the constraint very expressive. At the same time, we utilize a network simplex algorithm to design a highly efficient, and incremental, domain filtering algorithm. We thus integrate two powerful techniques for discrete optimization: constraint programming and the network simplex algorithm. Our generic constraint can be applied to automatically implement effective and efficient domain filterng algorithms for ad-hoc networks, but also for existing global constraints that rely on a network structure, including several soft global constraints many of which are not yet supported by CP systems. Our experimental results demonstrate the efficiency of our constraint, that can achieve speed-ups of several orders of magnitude with negligible overhead, when compared to a decomposition into primitive constraints.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints, CSP*; G.4 [**Mathematics of Computing**]: Mathematical software—*Algorithm design and analysis*

General Terms

Constraint programming, network flow constraint

Keywords

Minimum-cost network flow, global constraint

1. INTRODUCTION

Constraint programming (CP) is a paradigm to solve combinatorial optimization problems, combining an expressive modeling language with powerful inference techniques and systematic search. In addition to algebraic and logical relations over problem variables, CP allows the use of so-called *global constraints* that provide shorthands to often-used combinatorial substructures [6, 13]. Global constraints embed specialized inference techniques that exploit the associated combinatorial structure of the constraint, often allowing stronger levels of reasoning than would be possible with a decomposed representation of the combinatorial structure. Nowadays, global constraints are considered to be one of the most important components of CP systems in practice.

Many global constraints utilize a combinatorial structure that takes the form of a specific network flow. For example, the *alldifferent* constraint is equivalent to a bipartite matching problem [10], while the *global cardinality constraint*, or *gcc*, can be represented by a bipartite network flow [11]. Also when an objective function is combined with a global constraint, thus forming a so-called 'optimization constraint', the correspondence with network flows has been exploited. One well-known example is the *weighted gcc*, that corresponds to a weighted bipartite network flow [12].

Most, if not all, CP systems implement a dedicated inference algorithm for each individual global constraint. Even when global constraints are based on network flows, and share similar data structures and procedures, each specialized network flow algorithm would have to be implemented and tuned individually. The advantage of this approach is that CP systems can offer highly efficient algorithms for these constraints. In recent years, however, many more network-based global constraints have appeared in the literature, especially in the context of *soft* global constraints that can be applied to model and solve over-constrained problems [5]. This imposes a burden, as it may be unreasonable to implement and offer a dedicated algorithm for each such global constraint. In fact, most CP systems do not support any soft global constraint. A generic network flow constraint allows to simulate the performance of the specialized individual constraints in a convenient way.

Furthermore, many real-world problems contain a substructure that can be represented by some network flow, but not necessarily in the form of a global constraint that is offered by CP systems. Examples include process engineering, scheduling, routing, and transportation problems; see [16] for a discussion of constraint programming applications for networks. In order to exploit such 'ad-hoc' network flow structures, a generic network flow constraint would be very convenient, both as a modeling tool and as a propagation algorithm.

The main goal of this work is therefore to develop a *generic network flow constraint*. The purpose of this constraint is to $i)$ allow to model ad-hoc network flow structures, $ii)$ allow to model existing global constraints, most importantly soft global constraints, and $iii)$ embed efficient inference algorithms, providing the best trade-off between modeling power and practical solving efficiency.

Our first main contribution is the introduction of a more expressive generic Network Flow Constraint (*NFC*) than is currently available in the literature. Particular features of the *NFC* are that the following aspects of the network flow can be interpreted as a variable: the total (weighted) value of the flow, the flow on each arc, but also the unit cost of each arc. In addition, the constraint supports a 'structural' variable for each arc that can be used to enforce that the flow on the arc must equal its lower or upper capacity. Finally, we show how the node balance, node capacity, and node cost can be interpreted and handled as variables. These features allow us to model a wide range of network flow problems, including the application areas mentioned above.

The second contribution of this paper is the embedding of very efficient inference algorithms (in the form of so-called domain filtering algorithms), that are based on the network simplex method [1]. Most of the domain filtering algorithms for network flow-based global constraints, including *alldifferent*, *gcc*, and *soft-alldifferent*, apply some variant of the combinatorial 'successive shortest paths' network flow algorithm. Indeed, for these global constraints that are defined on bipartite networks, it can be shown that this approach yields powerful and provably efficient algorithms. For more general networks, however, these algorithms may no longer be the most efficient approach, most importantly because the worst-case pseudo-polynomial time complexity is often encountered in practice. Instead, the network simplex is among the most efficient network flow algorithms used in practical network flow applications. Moreover, as we will see, the network simplex method can be naturally embedded inside a propagation algorithm and backtrack-search environment. Since our main focus is on practical efficiency and re-usability, we propose to balance the amount of filtering and the associated computational load, resulting in highly efficient performance. In particular, we consider checking the consistency of a domain value only if our heuristic suggests it may be inconsistent and can be filtered from its domain.

The third contribution of this work is the application of our generic constraint to a number of network flow problems, including *soft global constraints*. We show that the *NFC* can be easily configured to represent, e.g., the *soft-alldifferent* constraint, and we experimentally demonstrate the computational performance of the *NFC* in practice. Interestingly, the *NFC* can obtain speed-ups of several orders of magnitude with respect to a corresponding decomposition of the problem, while the computational overhead is negligible.

Lastly, we have made our implementation of the *NFC* publicly available as an add-on to the open-source CP solver JaCoP [8]. We view this as an important contribution to the community, as it allows other researchers to quickly evaluate the performance of new (soft) global constraints based on network flows.

The remainder of the paper is organized as follows. In Section 2 we compare our generic network flow constraint with existing related work. Then, in Section 3 we provide necessary preliminaries on constraint programming and network flow theory. In Section 4 we introduce our generic network flow constraint. The associated propagation algorithm is described in Section 5. We present computational results in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Network flows play an important role in constraint programming, in particular because specialized network flow algorithms have been applied to design efficient filtering algorithms for several global constraints including *alldifferent* [10], *gcc* [11, 12], and their soft variants [7]. Bockmayr et al. were first to introduce a generic network flow constraint in a CP context [3]. Their constraint, called the *flow* constraint in [3], has been implemented in the CHIP solver. Similar to our *NFC*, the *flow* constraint can be applied to model minimum-cost network flow problems. Declaratively, our *NFC* is more generic in that all flow aspects can be interpreted as variables, whereas the *flow* constraint only considers as variable the total value of the flow, the arc flow, and the demand at each node. On the other hand, the *flow* constraint offers so-called 'conversion nodes' at the modeling level. These conversion nodes are not offered by the *NFC*, because they can be transformed into an equivalent multiple network flow problem (see Section 4). Indeed, this approach is also taken by the *flow* constraint to handle conversion nodes [3]. The domain filtering rules for the *flow* constraint are based on the computation of a network flow for feasibility reasoning, and on reduced costs for optimality reasoning. Our *NFC* also includes this reasoning as part of the propagation algorithm.

A different network flow constraint has been proposed in [2]; see also [4]. It can be used to model maximum flow problems (that is, no costs are involved), where the flow on each arc in the graph as well as the total flow value is represented by a finite domain variable. Therefore, this flow constraint is less generic than the *flow* constraint in [3] and our *NFC*. The domain filtering algorithm employs the highest-label preflow-push algorithm, which is particularly efficient for maximum flow problems.

Another closely related work is the *eplex* library [15], which provides a constraint programming interface to efficient linear programming techniques. Our *NFC* can be viewed as a network-specific variant of that work. Finally, in [9] it is discussed how global constraints such as *gcc*, *among*, and their combinations can be modeled in terms of a tractable set-intersection problem called 'Two Families of Sets' (TFOS). It is shown that domain filtering for the TFOS can be based on a maximum flow algorithm, and how the extension to the weighted TFOS problem can be represented by a minimum-cost flow.

3. PRELIMINARIES

3.1 Constraint Programming

We assume basic familiarity with constraint programming, and refer to [14] for more information.

A constraint programming problem is defined on a set of variables X, and a set of constraints C on subsets of X, and optionally an objective function $f : X \to \mathbb{Q}$. Each variable $x \in X$ has an associated finite domain $D(x)$. The goal is to find a variable assignment $x = d$ with $d \in D(x)$ for all $x \in X$ such that all constraints are satisfied, and f is optimized (if it exists).

For a set of variables X, we define $D(X) = \cup_{x \in X} D(x)$. We denote the lower and upper bound of a variable x by $x^{\min} = \min(D(x))$ and $x^{\max} = max(D(x))$, respectively.

3.2 The minimum-cost flow problem

We next present background information on network flow theory to fix terminology. For a more thorough treatment of network flow theory we refer to the textbook [1].

An instance of the minimum-cost flow problem on a directed graph is defined by a tuple (N, A, l, u, c, b), where

- N is the set of nodes,
- A is the set of directed arcs,
- $l : A \to \mathbb{Z}_{\geq 0}$ is the lower capacity function on the arcs,
- $u : A \to \mathbb{Z}_{\geq 0}$ is the upper capacity function on the arcs,
- $c : A \to \mathbb{Z}$ is the flow cost-per-unit function on the arcs,
- $b : N \to \mathbb{Z}$ is the node mass balance function on the nodes.

A *flow* is a function $x : A \to \mathbb{Z}_{\geq 0}$. The minimum-cost flow problem asks to find a flow that satisfies all arc capacity and node balance conditions, while minimizing total cost. It can be stated as follows:

$$\min \qquad z(x) = \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \qquad l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A, \tag{2}$$

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i \quad \forall i \in N \tag{3}$$

In addition to the parameters above, it is sometimes convenient to apply cost and capacity functions on the nodes as well:

- $l^n : N \to \mathbb{Z}_{\geq 0}$ is the lower capacity function on the nodes,
- $u^n : N \to \mathbb{Z}_{\geq 0}$ is the upper capacity function on the nodes,
- $c^n : N \to \mathbb{Z}$ is the flow cost-per-unit function on the nodes.

A network with cost and capacity functions on the nodes can be transformed into an equivalent network of the form (N, A, l, u, c, b) [1]. Namely, we can split a node $i \in N$ into two nodes i_{in} and i_{out} collecting all incoming, respectively outgoing arcs of i. We then introduce an arc $(i_{\text{in}}, i_{\text{out}})$ with arc cost c_i^n and lower and upper arc capacities l_i^n and u_i^n, respectively.

A flow x is *feasible* if it satisfies the arc capacity constraints (2) and node balance constraints (3). Under the assumption that a feasible flow exists, we can formulate the problem differently. For this formulation we need the concept of *residual network*. The residual network with respect to a feasible flow x is defined on the same node set N, but uses arc set $A_{\text{res}}(x)$ that is defined as follows. An arc (i, j) with flow x_{ij} has two copies in $A_{\text{res}}(x)$, arc (i, j) with residual lower capacity 0 and upper capacity $u_{ij} - x_{ij}$, and an arc (j, i) with residual lower capacity 0 and residual upper capacity $x_{ij} - l_{ij}$. Moreover, l_{ij} is subtracted from b_i and added to b_j (the residual node balances). Two arcs that correspond to the same edge in the network flow problem are called *sister arcs*.

In order to ensure that the network allows a feasible solution, we apply the standard technique of introducing auxiliary arcs that are used to potentially fulfill the node balance constraints, with associated penalty costs. If the original network has no solution, this is detected by the huge cost of the 'solution' in the extended network.

We introduce a 'potential' function $\pi : N \to \mathbb{Z}$ and define the *reduced cost* of an arc (i, j) as $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$. A feasible flow $x : A \to \mathbb{Z}_{\geq 0}$ is optimal if and only if there exists a potential function $\pi : N \to \mathbb{Z}$ such that:

$$c_{ij}^\pi \geq 0, \ \forall (i,j) \in A_{res}(x), \tag{4}$$

Algorithms for the minimum-cost flow can be separated into *primal* and *dual* algorithms depending on how they 'move' towards an optimal solution. Primal algorithms start with a feasible, but non-optimal flow and then iteratively improve the optimality of the flow until the optimality condition (4) is met. Conversely, dual algorithms start with an optimal, but infeasible solution. They iteratively improve the feasibility of the solution until the feasibility conditions (2) and (3) are met.

3.3 Network simplex algorithm

The network simplex algorithm uses the fundamental observation that if a minimum-cost network flow problem has a solution, then it has an optimal solution that can be represented as a spanning tree [1]. This is called a *spanning tree solution*. The basic idea of the algorithm is that at each step we move from one tree to another by replacing a tree arc with a non-tree arc, until we find an optimal spanning tree.

In a spanning tree solution, the flow x_{ij} of every arc (i, j) *not* in the spanning tree is either at its lower bound or at its upper bound. The network simplex algorithm maintains this information explicitly, by representing a spanning tree solution as the arc set partition (T, L, U), where T is the set of tree arcs, L is the set of arcs with flow $x_{ij} = l_{ij}$ and U is the set of arcs with flow $x_{ij} = u_{ij}$.

There are two basic operations defined on the spanning tree, called *primal pivot* and *dual pivot*. A primal pivot adds a non-tree arc to the spanning tree, thus creating a cycle in the tree. Flow is then sent through the cycle until some arc on the cycle reaches its upper capacity bound and is *blocking* the flow. This process is called *flow augmentation*. One of the blocking arcs is then removed from the tree structure, which eliminates the temporary cycle in the tree. In a *dual* pivot we first select the arc (i, j) that leaves the spanning tree. This will divide the set of nodes in two subsets (N_1, N_2), where each subset is reachable from exactly one endpoint i, j of the leaving arc, along the spanning tree. We then consider all arcs in the cut (N_1, N_2) and choose the arc with the lowest reduced cost to enter the tree and replace (i, j). After a primal or dual pivot we again have a spanning tree.

Even though the standard network simplex algorithm has an exponential worst-case time complexity in theory, it is well-known that this bound is virtually never encountered in practice. To the contrary; the method is known to be among the most efficient network flow algorithms in practice [1]. We will apply the primal algorithm to re-optimize the flow after (small) structural changes, while the dual algorithm will be applied to perform sensitivity analysis.

4. NETWORK FLOW CONSTRAINT

As stated before, the purpose of our network flow constraint is to be as expressive as possible, while being as computationally efficient as possible. The constraint definition provided below specifies which parts of the network flow are specified by variables and which parts of the network flow are fixed. Syntactically, we define our Network Flow Constraint as

$$\textit{NFC}(N, A, L, U, C, L^n, U^n, C^n, B, X, Z, S),$$

where the parameters are defined as follows:

- N is the list of nodes: fixed
- A is the list of directed arcs: fixed
- L is the list of lower capacities for the arcs: fixed
- U is the list of upper capacities for the arcs: fixed
- C is the list of unit costs for the arcs: fixed or variable
- L^n is the list of lower capacities for the nodes: fixed
- U^n is the list of upper capacities for the nodes: fixed
- C^n is the list of unit costs for the nodes: fixed or variable
- B is the list of mass balances for the nodes: fixed or variable
- X is the list of flow values for the arcs: fixed or variable
- Z is the total weighted flow value: variable
- S is the list of tuples providing structural rules on X and other problem variables

These parameters map directly (by capitalization) to the parameters that define the minimum-cost network flow problem in Section 3. We note that a user can define a maximization problem using the *NFC* by negating all costs of the network.

For internal representation and solving purposes, however, we only actively consider the variables in (C, X, Z, S). Namely, we can transform the network such that variables in B and C^n are represented by variables of type X and C. For node balance variables B, we introduce an artificial node n_0 and connect this node to all nodes i in N that have a node balance variable in B. For each such arc (i, n_0), the flow variable x_{i,n_0} is made equal to the corresponding variable b_i. The transformation of node cost variables C^n into arc cost variables C was already discussed in Section 3. We note that these transformations maintain the inference power of the constraint, without affecting its complexity.

All variable types C, X and Z are assumed to have an *interval* as their domain. That is, the propagation algorithm will only maintain and update the lower and upper bound of the domains.

The parameter S specifies a list of *structural rules* that are used to force the flow on an arc to be equal to its upper or lower capacity. A structural rule is specified by a tuple (v, d, a, t), where v is a finite domain variable, d is a domain (a finite set), a is an arc (i, j), and t denotes the type of the rule, being either UB (for upper bound) or LB (for lower bound). An UB rule is defined as

$$\begin{aligned} D(v) \subseteq d &\quad\Rightarrow\quad x_{ij} = u_{ij}, \\ x_{ij} < u_{ij} &\quad\Rightarrow\quad D(v) \cap d = \varnothing, \end{aligned} \tag{5}$$

while a LB rule is defined as

$$\begin{aligned} D(v) \cap d = \varnothing &\quad\Rightarrow\quad x_{ij} = l_{ij}, \\ x_{ij} > l_{ij} &\quad\Rightarrow\quad D(v) \cap d \neq \varnothing, \end{aligned} \tag{6}$$

where $x_{ij} \in X$, $l_{ij} \in L$, and $u_{ij} \in U$. These rules are specifically designed to map the network flow variables X to other problem variables v. The following example illustrates the use of structural rules when modeling the *soft-alldifferent* constraint using the *NFC*.

Example. Consider the constraint *soft-alldifferent*(X, z, μ_{dec}), where $X = \{x_1, \dots, x_n\}$ is a set of variables, z is a variable representing the total violation cost, and the decomposition-based violation measure μ_{dec} is defined as

$$\mu_{\text{dec}}(x_1, \dots, x_n) = |\{(i, j) \mid x_i = x_j, \text{ for } i < j\}|.$$

A tuple $(d_1, \dots, d_n, d)$ with $d_i \in D(x_i)$, $d \in D(z)$ is a solution to *soft-alldifferent*(X, z, μ_{dec}) if and only if $\mu_{\text{dec}}(d_1, \dots, d_n) \leq d$.

Solutions to the *soft-alldifferent* constraint can be represented by a minimum-cost network flow as follows [7]. We introduce a node set N_X representing the variables in X, and a node set N_D representing the values in $D(X)$. We also introduce a 'sink' node t. For each variable $x_i \in X$ and domain value $d \in D(x_i)$ we define an arc (x_i, d), as well as an arc (d, t). We refer to the arc set as A. Whenever d belongs to the domain of two or more variables, we have created parallel arcs from d to t. We order these parallel arcs in a fixed but arbitrary way, and define a cost $c_i = i - 1$ to the i-th arc from d to t, where $i = 1, 2, \dots$. The arcs from N_X to N_D have cost 0. Each arc a has a lower capacity $l_a = 0$ and upper capacity $u_a = 1$. We finally associate a node balance $b_{x_i} = 1$ to each node $x_i \in N_X$, while $b_t = -n$. An integer minimum-cost flow in the resulting graph has a one-to-one correspondence with a solution to the *soft-alldifferent* constraint minimizing the decomposition-based violation measure, by interpreting the unit flow on an arc (x_i, d) as the assignment $x_i = d$.

In order to model the *soft-alldifferent* constraint with the *NFC*, we need to map the assignments $x_i = d$ to the arcs (x_i, d), for all $x_i \in X$ and $d \in D(X)$. We do this by using structural rules, as follows. For each pair (x_i, d) we introduce both structural rules $(x_i, \{d\}, (x_i, d), \text{ UB})$ and $(x_i, \{d\}, (x_i, d), \text{ LB})$. Together, these will ensure that the flow on arc (x_i, d) is exactly 1 whenever $x_i = d$, and 0 otherwise.

The *soft-alldifferent* constraint can now be modeled using the *NFC* using the parameters described above. We remark that the parameters L^n, U^n and C^n are void in this case, and that z represents the total weighted flow variable Z in the *NFC*.

In addition to *soft-alldifferent*, we can apply the *NFC* to implement any existing network flow-based constraint, such as the classical (weighted and unweighted) *alldifferent* and *gcc* constraints, but also various other soft global constraints including the *soft-gcc* and the *soft-regular* constraints. We note that the current version of the *NFC* requires soft global constraints to have violation measures that can be expressed as convex costs on the arcs.

5. PROPAGATION ALGORITHM

5.1 Domain Filtering Algorithm

As discussed in Section 4, the *NFC* applies domain filtering (in fact, bounds reduction) to the three variable types C, X, Z, and the structural rules S. Domain filtering for the variables in S is performed by establishing domain consistency on the constraints that define the rules (5) and (6). We note that domain consistency is not guaranteed if for a particular rule v and x are the same variable. We next discuss the domain filtering with respect to Z, X and C.

Algorithm 1 represents the overall propagation algorithm of the *NFC*. The first goal is to verify that a solution exists, i.e, that the constraint is *consistent*. This is done by computing a minimum-cost flow, using the network simplex method; see line 2 of Algorithm 1. We note that only upon the first propagation event (usually at the root of the search tree), we need to run the network simplex algorithm from scratch. After that, the consistency check is performed incrementally by re-optimizing the network flow, taking into account the changes that have been made to the network structure. In particular we need to ensure that changes in the bounds of variables X, C, and Z still allow a feasible flow, with total cost at most $Z^{\max}$. We note that, when using the network simplex algorithm, the network flow can be re-optimized very quickly upon such changes. After (re-)optimization, we can immediately update the lower bound on the total flow value, i.e., $Z^{\min}$, as shown in line 4. Note that the *NFC* only ensures that $z(x) \leq Z^{\max}$, where $z(x)$ is the total cost of the flow (see equation (1)). Thus, the *NFC* prunes $Z^{\min}$ while other model constraints and search constraints prune $Z^{\max}$.

If no failure is thrown (line 6) then we perform a consistency analysis with respect to individual arcs of the network (line 9), in order to filter the domains of variables in X and C. The algorithm that analyzes a single arc is presented as Algorithm 2. For an arc (i, j) the goal of arc analysis is to find the maximum amount of flow through that arc, with cost at most $Z^{\max}$, corresponding to $x_{ij}^{\max}$ or $x_{ij}^{\min}$, depending on the orientation of the arc in the residual graph. Observe that in the residual network, the residual lower capacity l_{ij} for each arc (i, j) is 0.

To find the maximum amount of flow through an arc (i, j), we try to route as much flow from j to i as the remaining capacity of arc (i, j) allows for it, without using more than the remaining 'cost slack', as defined in line 8 of Algorithm 1. We can do this by first removing the arc (i, j) from the graph and then considering the

Algorithm 1 Main propagation

```
1: updateCache()
2: { cost, isFeasible } ← networkSimplex()
3: if ($isFeasible \wedge cost \leq Z^{\max}$) then
4:     $Z^{\min}$ ← max(cost, $Z^{\min}$)
5: else
6:     throw Failure
7: end if
8: costSlack ← $Z^{\max}$ - cost
9: arcAnalysis(costSlack)
```

Algorithm 2 Analyzing a single arc

```
1: { Input: (arc $(i, j)$, costSlack) }
2: source ← arc.head
3: sink ← arc.tail
4: capacity ← arc.capacity
5: flow ← 0
6: while ( capacity $> 0$ ) do
7:     unitCost ← $c^{\pi}_{ij}$
8:     if $(unitCost > 0)$ then
9:         maxCapacity ← costSlack / unitCost
10:        if ( capacity $>$ maxCapacity ) then
11:            capacity ← maxCapacity
12:            if ( capacity $= 0$) then
13:                break
14:            end if
15:        end if
16:    end if
17:    { delta, bArc } ← augmentFlow( source, sink, capacity )
18:    flow ← flow + delta
19:    capacity ← capacity - delta
20:    costSlack ← costSlack - unitCost * delta
21:    if ( capacity $= 0 \vee$!dualPivot( bArc )) then
22:        break
23:    end if
24: end while
25: if ( flow $<$ arc.capacity) then
26:     amount ← arc.capacity - flow
27:     if (arc.isForward) then
28:         $x^{\max}_{ij} \leftarrow x^{\max}_{ij} - amount$
29:     else
30:         $x^{\min}_{ij} \leftarrow x^{\min}_{ij} + amount$
31:     end if
32: end if
```

problem of sending u_{ij} units of flow from a source (j) to a sink (i), where u_{ij} represents the residual capacity of (i, j). This is done in several iterations. At each iteration we send one or more units of flow from j to i, by augmenting the flow along the j-i path in the spanning tree (there is exactly one path from j to i), i.e., here we apply primal pivot operations. We try to send the maximum amount of flow possible (line 17 of Algorithm 2) and by doing so we will either reach the capacity limit of the original arc (i, j) or create a *blocking arc* (called 'bArc') on the augmenting path. A blocking arc prevents us from sending more flow and has to be replaced by another arc with non-zero residual capacity. This is done by a dual pivot operation (line 21). This operation returns false if no such arc exists. If this is the case, then the current computed flow from j to i is maximal and we exit the loop.

Another termination condition is when the flow becomes too costly. The cost of sending one more unit of flow is equal to the reduced cost of the arc (c^{π}_{ij}). Note that sending flow along the spanning tree has always cost 0. Since we started from an optimal solution, c^{π}_{ij} is non-negative and it will increase at each iteration as the spanning tree changes. This exposes the (convex) cost structure of the arc, which we use for cost-based pruning by comparing it to the cost slack.

We can prune the domain of $x_{ij} \in X$ whenever the maximum flow that can be sent from j to i is less than the arc capacity (line 25). Depending on the orientation (forward/backward) of the arc either the upper or lower bound of x_{ij} is narrowed. For the variables in C, we apply a more passive filtering algorithm. First, observe that the *NFC* can never increase $c^{\min}$ for $c \in C$, but it may decrease $c^{\max}$. We decrease $c^{\max}$ whenever we can determine that the flow requirements on (i, j) make $c^{\max}$ inconsistent, that is, we apply the condition $c^{max}_{ij} \leq c^{min}_{ij} + \text{costSlack}/x^{min}_{ij}$.

The correctness of Algorithm 2 follows from standard network flow theory [1]. In fact, lines 6–7, 17–19, 21–24 implement a dual network simplex algorithm, while lines 8–16, 20 implement the cost-based pruning procedure.

5.2 Efficiency and Improvements

Performing an analysis for all arcs in the network would be quite expensive, most importantly because in many cases it may not result in any domain filtering. Therefore, we only consider those arcs that seem most promising to yield actual pruning of the variable domains. To this end, we employ a scoring heuristic to choose which arcs should be analyzed. Each arc is assigned a score that indicates how likely we are able to do successful pruning based on that arc. Here, successful pruning means that we have pruned the domain of one of the variables associated to that arc. During our propagation algorithm we only consider the top $P\%$ of arcs with the highest scores, for some percentage P.

Our dynamically updated scoring heuristic is based on collecting the *history* of an arc with respect to pruning. That is, if an arc yielded successful pruning, its score is increased. If an arc was analyzed but no associated variable domain could be pruned, then the score is decreased. We will see that the experiments support the assumption that the history of pruning is a relevant indicator of the pruning potential for the remaining part of the search.

For nodes that are connected to only 1 or 2 arcs we can achieve stronger pruning than Algorithm 2. If node $n_i \in N$ has degree 1 then there is only one possible flow assignment for the arc connected to that node. Depending on the direction of the arc we set the corresponding X-variable to $+b_i$ or $-b_i$, where b_i denotes the balance of n_i. In effect, the flow on the arc is then fixed, and it can be removed from the network. Next, consider the situation of a node n_i with degree 2. Let x_1 and x_2 be the flow variables of the two arcs connected to that node. The relation between x_1 and x_2 can be expressed as

$$d_1 \cdot x_1 = d_2 \cdot x_2 + b_i$$

where $d_i \in \{-1, 1\}$, $i = 1, 2$, depending on the direction of the arcs. We perform a standard domain consistency propagation algorithm for these binary constraints.

5.3 Implementation Details

We next discuss the most important data structures of the algorithm, as well as implementation details concerning efficiency, and state restoration upon backtracking.

Residual Network. We store the network in the form of a residual network. Recall that the residual network defines two sister arcs for each arc $(i, j) \in A$. In effect, we do not need to treat forward and residual (backward) arcs differently. In our implementation, sister arcs have pointers to each other. For efficiency we allow residual arcs to have upper capacity of 0, so that we do not have to allocate or de-allocate arcs when the flow is redistributed.

Spanning Tree. In order to maintain the spanning tree solution of the network simplex algorithm, we apply the parent-thread-depth (PTD) representation [1]. The PTD representation allows pre-order traversal in $O(1)$ memory because it stores the pre-order explicitly using pointers. Recall that the network simplex algorithm requires access to three sets of arcs (T, L, U). Our spanning tree already contains the tree arcs T, while we store the set of arcs at their lower bound (L) in a global list of arcs. The list of arcs at their upper bound (U) does not need to be maintained explicitly, since each arc in U is a sister arc of an arc in L.

We also utilize the information in (T, L, U) during the arc analysis. Namely, arcs (i, j) in L have a supporting flow for $x^{\min}_{ij}$, while arcs (i, j) in U have a supporting flow for $x^{\max}_{ij}$. Therefore, for these arcs the consistency of one bound comes for free, and we only perform the arc analysis for the other direction. Moreover, some arcs in T may be at their lower or upper bound, which again saves the check for one bound.

Caching. Often, certain parts of the flow are fixed and we could use an integer constant instead of singleton variables. Moreover, an efficient approach also takes advantage of the fact that variables change relatively rarely when compared to how often their state is being read. Therefore, the cached residual graph representation is only being updated if values of $x_{ij}^{\min}, x_{ij}^{\max}$ and $c_{ij}^{\min}$ change. Caching has a number of advantages and one disadvantage. First, if we do not cache, recomputing flow bounds of arcs in residual graph with the help of $min()$ and $max()$ functions is expensive when compared to one memory access. Caching results in a large performance boost since flow bounds access operations are performed extremely often in our algorithm. Moreover, for singleton variables the domain does no longer change, and we do not need to update the cache. However, one disadvantage is that the cache must be maintained (procedure $updateCache()$ in line 1 of Algorithm 1).

Backtracking. Upon backtracking, the *NFC* must restore its data to make it consistent with the search state. In order to do this efficiently and with little memory, we only remember the set of modified or deleted arcs at each search level. Their state can then be restored by refreshing the cached values with the state of the corresponding search variables. Note that we can reuse the current flow, the current node potentials (reduced costs) and the current spanning tree along with the associated arc sets (T, L, U). On backtracking, we only need to re-establish the invariants of the spanning tree solution when it restores the previous state. This will require at most a few pivot operations on the tree. As backtracking can only destroy the optimality of the flow, but not its feasibility, we normally would apply the primal algorithm to re-optimize the flow. However, since consistency also uses the primal algorithm we can simply defer the re-optimization until then and save execution time.

6. EXPERIMENTAL RESULTS

The main purpose of our experiments is to assess the computational efficiency of various aspects of the *NFC*, including the overall efficiency and the performance of the arc selection heuristic. In addition, we compare the computational efficiency of the *NFC* to decomposed models in which only primitive constraints are used, to evaluate the potential gain in performance when using a network flow-based global constraint.

6.1 Soft All-Different

Our first experiments are performed on problems that consist of a single decomposition-based *soft-alldifferent* constraint (as in the example of Section 4). The reason for using a single *soft-alldifferent* constraint is that domain consistency can be established in polynomial time for this constraint, which allows us to evaluate the quality of the arc scoring heuristic. Namely, we can increase the number of arcs to include until we effectively approach domain consistency.

We represent this problem using the *NFC*, and with a decomposed model using primitive constraints. The decomposition associates a Boolean variable t_{ij} to each equality constraint $x_i = x_j$, yielding the reified constraint $t_{ij} \Leftrightarrow (x_i = x_j)$, for all $i < j$. The total violation is expressed as $z = \sum_{i<j} t_{ij}$.

We created random problem instances based on three parameters, N, *MaxL* (for maximum length), and *MaxC* (for maximum cost). They define the constraint *soft-alldifferent*(X, z, μ_{dec}) where $|X| = N$ and $D(z) = [0, \textit{MaxC}]$. The domains of the variables in

	# Arcs considered	% Arcs pruned	# Nodes	# Wrong decisions	Time (s)
Primitive	-	-	333,147	162,014	4.0
NFC	0	-	118,371	54,626	9.12
NFC	1	25.8	23,395	7,138	2.29
NFC	2	10.7	13,679	2,280	1.82
NFC	3	10.6	11,233	1,057	1.81
NFC	4	9.1	9,907	394	1.89
NFC	5	8.2	9,327	104	1.9
NFC	6	7.5	9,239	60	2.1
NFC	7	6.5	9,195	38	2.13
NFC	8	5.9	9,143	12	2.15
NFC	9	5.7	9,141	11	2.19
NFC	10	5.4	9,137	9	2.2
NFC	11	5.2	9,125	3	2.28
NFC	12	5.1	9,123	2	2.29
NFC	13	5.0	9,121	1	2.32
NFC	14	5.0	9,121	1	2.34
NFC	29	4.8	9,119	0	2.4

Table 1: The *NFC* applied to a *soft-alldifferent* problem with a *sparse* solution space of only 9,120 solutions. The parameters are N = 19, *MaxL* = 4 and *MaxC* = 3; the network has 38 nodes and 390 arcs. We search for all solutions. The *NFC* is configured to only prune a small number of arcs per search node.

X are defined as follows. Let $random(m)$ be a function that produces a random number uniformly chosen from the interval $[1, m]$. For each variable x_i, we generate $len_i = random(\textit{MaxL} - 1)$ and $min_i = random(n - len_i)$. We then define $D(x_i) = [min_i, (min_i + len_i)]$. Note that the size of $D(x_i)$ is $len_i + 1$ and lies between 2 and *MaxL*. We consider various problem variants to these instances: finding all solutions, proving unsatisfiability (that is, there exists no assignment for X such that $z \leq \textit{MaxC}$, and finding a minimum-cost solution.

We first investigate the performance of our arc scoring heuristic. Recall that at each propagation event we only perform arc analysis on a subset of the arcs in the network, having the largest score. Tables 1 and 2 show the results on problems with different characteristics. The first problem has a sparse solution space of only 9,120 solutions, while the second problem is much less constrained and allows 54,996 solutions. In both tables, we find all solutions to the problem. In these tables, we report the performance of the *NFC* when the number of arcs considered is increased. Here (% Arcs pruned') refers to the average success ratio of pruning, i.e. how often arc analysis leads to actual pruning, given by

$$\text{\% of arcs pruned} = \frac{\text{\# of arcs pruned}}{\text{\# of arcs examined}}.$$

We can observe that the number of wrong decisions is roughly divided by 2 for every additional arc that we attempt to prune. Interestingly, the results show that when only 29 (Table 1), respectively 14 (Table 2), arcs are considered during each filtering event (about 7.4%, resp. 6.0%, of the total number of arcs), the corresponding pruning is sufficient to make no wrong decisions during search (similar to what domain consistency would achieve). This suggests that our scoring heuristic is performing quite well.

These tables also illustrate the impact of solution density, that can be adjusted with the parameter *MaxC*. When the solution space is smaller (Table 1), the *NFC* has more pruning opportunities, and can outperform the decomposition using primitive constraints. However, when the solution space is larger (Table 2), the additional pruning may not lead to a reduced computation time when only a single constraint is considered.

	# Arcs considered	% Arcs pruned	# Wrong decisions	Time (s)
Primitive	-	-	32,932	1.3
NFC	0	-	63,592	6.9
NFC	1	19.4	26,838	7.4
NFC	2	13.6	16,146	8.0
NFC	3	12.2	8,818	8.2
NFC	4	11.2	4,137	8.1
NFC	5	10.6	1,934	8.5
NFC	6	9.9	718	8.9
NFC	7	9.3	384	8.9
NFC	8	9.1	199	9.0
NFC	9	8.7	45	9.5
NFC	10	8.3	30	9.3
NFC	11	8.3	10	9.4
NFC	12	8.1	4	9.5
NFC	13	8.0	7	9.6
NFC	14	8.0	0	9.6

Table 2: The *NFC* applied to a *soft-alldifferent* problem with a *dense* solution space with 54,996 solutions. The parameters are *N* = 15, *MaxL* = 4 and *MaxC* = 5; the network has 29 nodes and 232 arcs. We search for all solutions.

	N	*MaxL*	*MaxC*	# Nodes	# Wrong Decisions	Time (s)
NFC	17	4	2	0	0	0.023
Primitive	17	4	2	8951	4476	0.164
NFC	20	7	3	0	0	0.04
Primitive	20	7	3	840,011	420,006	8.34
NFC	24	4	6	0	0	0.024
Primitive	24	4	6	$>$4,372,355	$>$2,186,170	$>$60

Table 3: Unsatisfiable instances of *soft-alldifferent*. The *NFC* fails immediately without search. Primitive constraints time out on the last instance.

In order to further investigate the power of the *NFC*, we next consider unsatisfiable problem instances, for which *MaxC* is too low to allow any solution. These instances expose an important difference between the *NFC* and the decomposition: the *NFC* can immediately deduce that an instance is unsatisfiable, while primitive constraints cannot, as illustrated in Table 3. All instances in this table are unsatisfiable. The *NFC* fails immediately at the first node, whereas primitive constraints need to explore a large tree to prove that the instance is unsatisfiable.

Finally, we compare the *NFC* and the decomposition on the optimization variant of this problem, in which we want to find a solution with minimum total violation cost. The optimization in the search is achieved by decreasing the upper bound of the cost variable every time a solution is found. The search stops when no more solutions can be found. As there may be many suboptimal (infeasible) search nodes, it is expected that the *NFC* performs better than the decomposition into primitive constraints, based on the results in Table 3. The performance of the different models is presented in Table 4. The reported instances in this table all have at least one feasible solution. Observe that the primitive constraints indeed perform poorly in this case. They are not aware of the optimization goal, and the upper limit on the cost has little direct influence on the propagation. The performance of *NFC* is remarkable, as already for small problems *NFC* uses four orders of magnitude less search nodes than the decomposition.

	N	*MaxL*	Cost	# Nodes	Time (s)
NFC	17	5	1	42	0.154
Primitive	17	5	1	896,655	10.4
NFC	20	5	1	44	0.155
Primitive	20	5	(2)	1,569,962	>30
NFC	50	5	4	239	2.4
Primitive	50	5	(10)	1,270,823	>30
NFC	70	5	8	605	11.4
Primitive	70	5	(25)	647,708	>30
NFC	17	10	0	88	0.154
Primitive	17	10	0	55,068	0.685
NFC	20	10	3	64	0.155
Primitive	20	10	(4)	2,175,059	>30
NFC	50	10	8	285	4.8
Primitive	50	10	(18)	947,174	>30
NFC	70	10	8	723	21.5
Primitive	70	10	(32)	382,176	>30

Table 4: Finding an optimal solution for *soft-alldifferent*. Primitive constraints time out for the network with 20 or more variables. A cost in parenthesis is the minimal cost that was found before timeout.

6.2 Personnel Scheduling

The next set of experiments is performed on a personnel scheduling problem introduced in [3]. In this problem, we need to assign 8-hour shifts to telephone operators. A day is divided into 6 periods of 4 hours. Each period has a minimum requirement on the number of operators. We assume that operators work for a consecutive period of 8 hours and that they can start to work at the beginning of any of the 6 periods. The objective is to minimize the number of shifts while respecting the minimum requirements for each period.

The personnel scheduling problem has been used by [3] to test the implementation of their network flow constraint. Unfortunately, they only publish their model but not their benchmarking results. We will still use the same data set that they proposed: the minimum requirement of working operators per each shift are $\{26, 52, 86, 120, 75, 35\}$. This problem can be modeled using a simple network with 6 nodes and 12 arcs. Nodes correspond to the beginning of a time period. There are two types of arcs: *working arcs* going from t to $t + 4$ hours and *free arcs* lasting from t to $t + 16$ hours (both modulo 24 hours). Working arcs have a cost of one per unit of flow and a lower capacity equal to minimum requirements for that period. Free arcs have no cost and a lower capacity of zero.

For this problem, the decomposed model consists of the individual constraints (2) and (3) that constitute the network flow problem of Section 3. In our experiments we let both approaches, the *NFC* and the primitive model, find all solutions up to a given cost bound. We measure the time and the number of search nodes for each executions. The results, shown in Table 5 show that the *NFC* makes no wrong decisions. Interestingly, there is virtually no overhead in using the *NFC* over the primitive model. Both process nodes at about the same rate. Together, the *NFC* is faster by 11x (cost 425) to 19x (cost 455) when compared to the model using primitive constraints.

6.3 Random Shift Scheduling Networks

The *NFC* was designed to be very expressive, but it is difficult to find benchmark problems that use all properties of our constraint simultaneously. Therefore, in this last set of experiments, we apply random networks to evaluate all the *NFC* features simultaneously. We compare the *NFC* against a decomposed model that applies global sum and weighted global sum constraints that constitute the

	Maximal cost	# Solutions	# Search Nodes	Time (s)	# Nodes / s
NFC	415	231	230	0.13	1.77
Primitive	415	231	5,650	0.47	12.0
NFC	425	6,496	6,495	0.57	11.4
Primitive	425	6,496	75,607	6.23	12.1
NFC	435	26,411	26,410	1.92	13.8
Primitive	435	26,411	351,198	27.8	12.8
NFC	445	68,460	68,459	4.68	14.6
Primitive	445	68,460	1,112,925	76.7	14.5
NFC	455	141,960	141,959	9.0	15.8
Primitive	455	141,960	2,784,763	171	16.3

Table 5: Finding all operator schedules up to a given cost. The *NFC* mimics domain consistency (no wrong decisions) on all cases, while processing nodes at the same rate as primitive constraints.

	# Arcs	# Solutions	# Search Nodes	# Wrong Decisions	Time (s)
NFC	18	4,712	48,035	21,662	0.29
Primitive	18	4,712	51,067	23,178	5.57
NFC	20	621	9,288	4,334	0.19
Primitive	20	621	44,068	21,724	9.67
NFC	30	4,587	34,882	15,148	0.28
Primitive	30	4,587	52,638	24,026	10.1
NFC	40	35,123	478,662	221,770	4.14
Primitive	40	(0)	265,469	132,730	>120

Table 6: Finding all solutions of randomly generated shift scheduling problems.

network flow problem.

The random networks have a cyclical structure similar to the personnel scheduling problem in the previous section, but with more shifts, and with variable arc costs. Each node has an arc to the next node in the cycle and to two nodes earlier in the cycle. Our instances with m arcs contain $m/2$ shifts (each shift is represented by a node). Each arc (i, j) has associated variables $x_{ij} \in X$ and $c_{ij} \in C$. For both variable types, the domains are generated randomly as an interval between *min* and *max*. Here *min* is a number produced by $random(10) - 1$, yielding the range $[0, 9]$, while *max* is equal to $21 - random(10)$ yielding the range $[11, 20]$. Recall that $random(m)$ produces a random number uniformly chosen from the interval $[1, m]$.

Table 6 shows the results that we obtained for instances with 18, 20, 30, and 40 arcs. Observe that the *NFC* is performing more pruning, resulting in fewer search nodes, than the decomposition. More importantly, however, the *NFC* is at least one order of magnitude faster already for small problems. Realizing that the global sum constraints of the decomposed model are already optimized, the difference in efficiency is even more striking. Apparently, the decomposition not only loses the network structure, but it also pays a computational price in the overhead of maintaining a large number of smaller constraints.

7. CONCLUSIONS

We have introduced a generic network flow constraint that can be applied to model ad-hoc global constraints for network problems as well as existing global constraints that utilize a specific network flow representation, in particular soft global constraints. We have shown how the efficient network simplex algorithm can be embedded inside the domain filtering filtering associated with this network flow constraint. We have evaluated the performance of our network flow constraint, and demonstrated its efficiency with respect to corresponding decomposed models, on three application areas: soft global constraints, realistic network flow problems (personnel scheduling), and randomly generated shift scheduling problems exhibiting all features modeled by our constraint. Our experiments indicate that the network flow constraint can achieve speedups of orders of magnitude compared to a corresponding decomposition into primitive constraints.

8. REFERENCES

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows - Theory, Algorithms and Applications*. Prentice-Hall, 1993.

[2] T. Benoist, E. Gaudin, and B. Rottembourg. Constraint Programming Contribution to Benders Decomposition: A Case Study. In *Proceedings of CP*, volume 2470 of *LNCS*, pages 603–617. Springer, 2002.

[3] A. Bockmayr, N. Pisaruk, and A. Aggoun. Network Flow Problems in Constraint Programming. In *Proceedings of CP*, volume 2239 of *LNCS*, pages 196–210. Springer, 2001.

[4] É. Gaudin, N. Jussien, and G. Rochart. Implementing explained global constraints. In *Proceedings of the CP'04 Workshop on Constraint Propagation and Implementation*, pages 61–76, 2004.

[5] W.-J. van Hoeve. Over-Constrained Problems. In P. Van Hentenryck and M. Milano, editors, *Hybrid Optimization: the 10 years of CPAIOR*, chapter 6. Springer, 2010.

[6] W.-J. van Hoeve and I. Katriel. Global constraints. In Rossi et al. [14], chapter 6.

[7] W.-J. van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4):347–373, 2006.

[8] Java Constraint Programing (JaCoP) solver. http://jacop.osolpro.com.

[9] I. Razgon, B. O'Sullivan, and G. Provan. Generalizing Global Constraints Based on Network Flows. In *Recent Advances in Constraints*, volume 5129 of *LNCS*, pages 127–141. Springer, 2008.

[10] J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of AAAI*, volume 1, pages 362–367. AAAI Press, 1994.

[11] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of AAAI/IAAI*, volume 1, pages 209–215. AAAI Press/The MIT Press, 1996.

[12] J.-C. Régin. Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, 7(3-4):387–405, 2002.

[13] J.-C. Régin. Global Constraints and Filtering Algorithms. In M. Milano, editor, *Constraint and Integer Programming - Toward a Unified Methodology*, chapter 4. Kluwer Academic Publishers, 2003.

[14] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[15] K. Shen and J. Schimpf. Eplex: Harnessing Mathematical Programming Solvers for Constraint Logic Programming. In *Proceedings of CP*, volume 3709 of *LNCS*, pages 622–636. Springer, 2005.

[16] H. Simonis. Constraint applications in networks. In Rossi et al. [14], chapter 25.